# W3School C# & Asp.net 教程

wizardforcel

Published with GitBook



# 目錄

# W3School C# & ASP.net教程

来源：

- C#教程
- ASP.net教程

整理：飞龙

# C# 基础

# C# 简介

C# 是一个现代的、通用的、面向对象的编程语言，它是由微软（Microsoft）开发的，由 Ecma 和 ISO 核准认可的。

C# 是由 Anders Hejlsberg 和他的团队在 .Net 框架开发期间开发的。

C# 是专为公共语言基础结构（CLI）设计的。CLI 由可执行代码和运行时环境组成，允许在不同的计算机平台和体系结构上使用各种高级语言。

下面列出了 C# 成为一种广泛应用的专业语言的原因：

- 现代的、通用的编程语言。
- 面向对象。
- 面向组件。
- 容易学习。
- 结构化语言。
- 它产生高效率的程序。
- 它可以在多种计算机平台上编译。
- .Net 框架的一部分。

# C# 强大的编程功能

虽然 C# 的构想十分接近于传统高级语言 C 和 C++，是一门面向对象的编程语言，但是它与 Java 非常相似，有许多强大的编程功能，因此得到广大程序员的亲睐。

下面列出 C# 一些重要的功能：

- 布尔条件（Boolean Conditions）
- 自动垃圾回收（Automatic Garbage Collection）
- 标准库（Standard Library）
- 组件版本（Assembly Versioning）
- 属性（Properties）和事件（Events）
- 委托（Delegates）和事件管理（Events Management）
- 易于使用的泛型（Generics）
- 索引器（Indexers）
- 条件编译（Conditional Compilation）
- 简单的多线程（Multithreading）
- LINQ 和 Lambda 表达式
- 集成 Windows

# C# 环境

在这一章中，我们将讨论创建 C# 编程所需的工具。我们已经提到 C# 是 .Net 框架的一部分，且用于编写 .Net 应用程序。因此，在讨论运行 C# 程序的可用工具之前，让我们先了解一下 C# 与 .Net 框架之间的关系。

# .Net 框架（.Net Framework）

.Net 框架是一个创新的平台，能帮您编写出下面类型的应用程序：

- Windows 应用程序
- Web 应用程序
- Web 服务

.Net 框架应用程序是多平台的应用程序。框架的设计方式使它适用于下列各种语言：C#、C++、Visual Basic、Jscript、COBOL 等等。所有这些语言可以访问框架，彼此之间也可以互相交互。

.Net 框架由一个巨大的代码库组成，用于 C# 等客户端语言。下面列出一些 .Net 框架的组件：

- 公共语言运行库（Common Language Runtime - CLR）
- .Net 框架类库（.Net Framework Class Library）
- 公共语言规范（Common Language Specification）
- 通用类型系统（Common Type System）
- 元数据（Metadata）和组件（Assemblies）
- Windows 窗体（Windows Forms）
- ASP.Net 和 ASP.Net AJAX
- ADO.Net
- Windows 工作流基础（Windows Workflow Foundation - WF）
- Windows 显示基础（Windows Presentation Foundation）
- Windows 通信基础（Windows Communication Foundation - WCF）
- LINQ

如需了解每个组件的详细信息，请参阅微软（Microsoft）的文档。

# C# 的集成开发环境（Integrated Development Environment - IDE）

微软（Microsoft）提供了下列用于 C# 编程的开发工具：

- Visual Studio 2010 (VS)
- Visual C# 2010 Express (VCE)
- Visual Web Developer

后面两个是免费使用的，可从微软官方网址下载。使用这些工具，您可以编写各种 C# 程序，从简单的命令行应用程序到更复杂的应用程序。您也可以使用基本的文本编辑器（比如 Notepad）编写 C# 源代码文件，并使用命令行编译器（.NET 框架的一部分）编译代码为组件。

Visual C# Express 和 Visual Web Developer Express 版本是 Visual Studio 的定制版本，且具有相同的外观和感观。它们保留 Visual Studio 的大部分功能。在本教程中，我们使用的是 Visual C# 2010 Express。

您可以从 Microsoft Visual Studio 上进行下载。它会自动安装在您的机器上。请注意，您需要一个可用的网络连接来完成速成版的安装。

# 在 Linux 或 Mac OS 上编写 C# 程序

虽然 .NET 框架是运行在 Windows 操作系统上，但是也有一些运行于其它操作系统上的版本可供选择。**Mono** 是 .NET 框架的一个开源版本，它包含了一个 C# 编译器，且可运行于多种操作系统上，比如各种版本的 Linux 和 Mac OS。如需了解更多详情，请访问 Go Mono。

Mono 的目的不仅仅是跨平台地运行微软 .NET 应用程序，而且也为 Linux 开发者提供了更好的开发工具。Mono 可运行在多种操作系统上，包括 Android、BSD、iOS、Linux、OS X、Windows、Solaris 和 UNIX。

# C# 程序结构

在我们学习 C# 编程语言的基础构件块之前，让我们先看一下 C# 的最小的程序结构，以便作为接下来章节的参考。

# C# Hello World 实例

一个 C# 程序主要包括以下部分：

- 命名空间声明（Namespace declaration）
- 一个 class
- Class 方法
- Class 属性
- 一个 Main 方法
- 语句（Statements）& 表达式（Expressions）
- 注释

让我们看一个可以打印出 "Hello World" 的简单的代码：

```
using System;
namespace HelloWorldApplication
{
   class HelloWorld
   {
      static void Main(string[] args)
      {
         /* 我的第一个 C# 程序*/
         Console.WriteLine("Hello World");
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Hello World
```

让我们看一下上面程序的各个部分：

- 程序的第一行 **using System; - using** 关键字用于在程序中包含 **System** 命名空间。 一个程序一般有多个 **using** 语句。
- 下一行是 **namespace** 声明。一个 **namespace** 是一系列的类。*HelloWorldApplication* 命名空间包含了类 *HelloWorld*。
- 下一行是 **class** 声明。类 *HelloWorld* 包含了程序使用的数据和方法声明。类一般包含多个方法。方法定义了类的行为。在这里，*HelloWorld* 类只有一个 **Main** 方法。

- 下一行定义了 **Main** 方法，是所有 C# 程序的 入口点。**Main** 方法说明当执行时 类将做什么动作。
- 下一行 /.../ 将会被编译器忽略，且它会在程序中添加额外的 注释。
- Main 方法通过语句 **Console.WriteLine("Hello World");** 指定了它的行为。

  *WriteLine* 是一个定义在 *System* 命名空间中的 *Console* 类的一个方法。该语句会在屏幕上显示消息 "Hello, World!"。

- 最后一行 **Console.ReadKey();** 是针对 VS.NET 用户的。这使得程序会等待一个按键的动作，防止程序从 Visual Studio .NET 启动时屏幕会快速运行并关闭。

以下几点值得注意：

- C# 是大小写敏感的。
- 所有的语句和表达式必须以分号（;）结尾。
- 程序的执行从 Main 方法开始。
- 与 Java 不同的是，文件名可以不同于类的名称。

# 编译 & 执行 C# 程序

如果您使用 Visual Studio.Net 编译和执行 C# 程序，请按下面的步骤进行：

- 启动 Visual Studio。
- 在菜单栏上，选择 File -> New -> Project。
- 从模板中选择 Visual C#，然后选择 Windows。
- 选择 Console Application。
- 为您的项目制定一个名称，然后点击 OK 按钮。
- 新项目会出现在解决方案资源管理器（Solution Explorer）中。
- 在代码编辑器（Code Editor）中编写代码。
- 点击 Run 按钮或者按下 F5 键来运行程序。会出现一个命令提示符窗口（Command Prompt window），显示 Hello World。

您也可以使用命令行代替 Visual Studio IDE 来编译 C# 程序：

- 打开一个文本编辑器，添加上面提到的代码。
- 保存文件为 **helloworld.cs**。
- 打开命令提示符工具，定位到文件所保存的目录。
- 键入 **csc helloworld.cs** 并按下 enter 键来编译代码。
- 如果代码没有错误，命令提示符会进入下一行，并生成 **helloworld.exe** 可执行文件。
- 接下来，键入 **helloworld** 来执行程序。
- 您将看到 "Hello World" 打印在屏幕上。

# C# 基本语法

C# 是一种面向对象的编程语言。在面向对象的程序设计方法中，程序由各种相互交互的对象组成。相同种类的对象通常具有相同的类型，或者说，是在相同的 class 中。

例如，以 Rectangle（矩形）对象为例。它具有 length 和 width 属性。根据设计，它可能需要接受这些属性值、计算面积和显示细节。

让我们来看看一个 Rectangle（矩形）类的实现，并借此讨论 C# 的基本语法：

```
using System;
namespace RectangleApplication
{
    class Rectangle
    {
        // 成员变量
        double length;
        double width;
        public void Acceptdetails()
        {
            length = 4.5;
            width = 3.5;
        }
        public double GetArea()
        {
            return length * width;
        }
        public void Display()
        {
            Console.WriteLine("Length: {0}", length);
            Console.WriteLine("Width: {0}", width);
            Console.WriteLine("Area: {0}", GetArea());
        }
    }

    class ExecuteRectangle
    {
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle();
            r.Acceptdetails();
            r.Display();
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Length: 4.5
Width: 3.5
Area: 15.75
```

# *using* 关键字

在任何 C# 程序中的第一条语句都是：

```
using System;
```

**using** 关键字用于在程序中包含命名空间。一个程序可以包含多个 using 语句。

## *class* 关键字

**class** 关键字用于声明一个类。

## C# 中的注释

注释是用于解释代码。编译器会忽略注释的条目。在 C# 程序中，多行注释以 / 开始，并以字符 / 终止，如下所示：

```
/* This program demonstrates
The basic syntax of C# programming
Language */
```

单行注释是用 '//' 符号表示。例如：

```
}//end class Rectangle
```

## 成员变量

变量是类的属性或数据成员，用于存储数据。在上面的程序中，*Rectangle* 类有两个成员变量，名为 *length* 和 *width*。

## 成员函数

函数是一系列执行指定任务的语句。类的成员函数是在类内声明的。我们举例的类 Rectangle 包含了三个成员函数： *AcceptDetails*、*GetArea* 和 *Display*。

## 实例化一个类

在上面的程序中，类 *ExecuteRectangle* 是一个包含 *Main()* 方法和实例化 *Rectangle* 类的类。

# 标识符

标识符是用来识别类、变量、函数或任何其它用户定义的项目。在 C# 中，类的命名必须遵循如下基本规则：

- 标识符必须以字母开头，后面可以跟一系列的字母、数字（0 - 9）或下划线（ _ ）。标识符中的第一个字符不能是数字。
- 标识符必须不包含任何嵌入的空格或符号，比如 ? - +! @ # % ^ & * ( ) [ ] { } . ; : " ' / \。但是，可以使用下划线（ _ ）。
- 标识符不能是 C# 关键字。

# C# 关键字

关键字是 C# 编译器预定义的保留字。这些关键字不能用作标识符，但是，如果您想使用这些关键字作为标识符，可以在关键字前面加上 @ 字符作为前缀。

在 C# 中，有些标识符在代码的上下文中有特殊的意义，如 get 和 set，这些被称为上下文关键字（contextual keywords）。

下表列出了 C# 中的保留关键字（Reserved Keywords）和上下文关键字（Contextual Keywords）：

| 保留关键字 | | | | | | |
|---|---|---|---|---|---|---|
| abstract | as | base | bool | break | byte | case |
| catch | char | checked | class | const | continue | decin |
| default | delegate | do | double | else | enum | even |
| explicit | extern | false | finally | fixed | float | for |
| foreach | goto | if | implicit | in | in (generic modifier) | int |
| interface | internal | is | lock | long | namespace | new |
| null | object | operator | out | out (generic modifier) | override | parar |
| private | protected | public | readonly | ref | return | sbyte |
| sealed | short | sizeof | stackalloc | static | string | struc |
| switch | this | throw | true | try | typeof | uint |
| ulong | unchecked | unsafe | ushort | using | virtual | void |
| volatile | while | | | | | |
| 上下文关键字 | | | | | | |
| add | alias | ascending | descending | dynamic | from | get |
| global | group | into | join | let | orderby | partia (type |
| partial (method) | remove | select | set | | | |

# C# 数据类型

在 C# 中，变量分为以下几种类型：

- 值类型（Value types）
- 引用类型（Reference types）
- 指针类型（Pointer types）

## 值类型（Value types）

值类型变量可以直接分配给一个值。它们是从类 **System.ValueType** 中派生的。

值类型直接包含数据。比如 **int**、**char**、**float**，它们分别存储数字、字母、浮点数。当您声明一个 **int** 类型时，系统分配内存来存储值。

下表列出了 C# 2010 中可用的值类型：

| 类型 | 描述 | 范围 | 默认值 |
|---|---|---|---|
| bool | 布尔值 | True 或 False | False |
| byte | 8 位无符号整数 | 0 到 255 | 0 |
| char | 16 位 Unicode 字符 | U +0000 到 U +ffff | '\0' |
| decimal | 128 位精确的十进制值，28-29 有效位数 | (-7.9 x $10^{28}$ 到 7.9 x $10^{28}$) / $10^{0}$ 到 28 | 0.0M |
| double | 64 位双精度浮点型 | (+/-)5.0 x $10^{-324}$ 到 (+/-)1.7 x $10^{308}$ | 0.0D |
| float | 32 位单精度浮点型 | -3.4 x $10^{38}$ 到 + 3.4 x $10^{38}$ | 0.0F |
| int | 32 位有符号整数类型 | -2,147,483,648 到 2,147,483,647 | 0 |
| long | 64 位有符号整数类型 | -923,372,036,854,775,808 到 9,223,372,036,854,775,807 | 0L |
| sbyte | 8 位有符号整数类型 | -128 到 127 | 0 |
| short | 16 位有符号整数类型 | -32,768 到 32,767 | 0 |
| uint | 32 位无符号整数类型 | 0 到 4,294,967,295 | 0 |
| ulong | 64 位无符号整数类型 | 0 到 18,446,744,073,709,551,615 | 0 |
| ushort | 16 位无符号整数类型 | 0 到 65,535 | 0 |

如需得到一个类型或一个变量在特定平台上的准确尺寸，可以使用 **sizeof** 方法。表达式 *sizeof(type)* 产生以字节为单位存储对象或类型的存储尺寸。下面举例获取任何机器上 *int* 类型的存储尺寸：

```
namespace DataTypeApplication
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine("Size of int: {0}", sizeof(int));
         Console.ReadLine();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Size of int: 4
```

# 引用类型（Reference types）

引用类型不包含存储在变量中的实际数据，但它们包含对变量的引用。

换句话说，它们指的是一个内存位置。使用多个变量时，引用类型可以指向一个内存位置。如果内存位置的数据是由一个变量改变的，其他变量会自动反映这种值的变化。内置的 引用类型有：**object**、**dynamic** 和 **string**。

## 对象（Object）类型

对象（**Object**）类型 是 C# 通用类型系统（Common Type System - CTS）中所有数据类型的终极基类。Object 是 System.Object 类的别名。所以对象（Object）类型可以被分配任何其他类型（值类型、引用类型、预定义类型或用户自定义类型）的值。但是，在分配值之前，需要先进行类型转换。

当一个值类型转换为对象类型时，则被称为 装箱；另一方面，当一个对象类型转换为值类型时，则被称为 拆箱。

```
object obj;
obj = 100; // 这是装箱
```

## 动态（Dynamic）类型

您可以存储任何类型的值在动态数据类型变量中。这些变量的类型检查是在运行时发生的。

声明动态类型的语法：

```
dynamic <variable_name> = value;
```

例如：

```
dynamic d = 20;
```

动态类型与对象类型相似，但是对象类型变量的类型检查是在编译时发生的，而动态类型变量的类型检查是在运行时发生的。

## 字符串（String）类型

字符串（**String**）类型 允许您给变量分配任何字符串值。字符串（String）类型是 System.String 类的别名。它是从对象（Object）类型派生的。字符串（String）类型的值可以通过两种形式进行分配：引号和 @引号。

例如：

```
String str = "w3cschool.cc";
```

一个 @引号字符串：

```
@"w3cschool.cc";
```

C# string 字符串的前面可以加 @（称作"逐字字符串"）将转义字符（\）当作普通字符对待，比如：

```
string str = @"C:\Windows";
```

等价于：

```
string str = "C:\\Windows";
```

@ 字符串中可以任意换行，换行符及缩进空格都计算在字符串长度之内。

```
string str = @"<script type=""text/javascript"">
    <!--
    -->
    </script>";
```

用户自定义引用类型有：class、interface 或 delegate。我们将在以后的章节中讨论这些类型。

# 指针类型（Pointer types）

指针类型变量存储另一种类型的内存地址。C# 中的指针与 C 或 C++ 中的指针有相同的功能。

声明指针类型的语法：

```
type* identifier;
```

例如：

```
char* cptr;
int* iptr;
```

我们将在章节"不安全的代码"中讨论指针类型。

# C# 类型转换

类型转换从根本上说是类型铸造，或者说是把数据从一种类型转换为另一种类型。在 C# 中，类型铸造有两种形式：

- 隐式类型转换 - 这些转换是 C# 默认的以安全方式进行的转换。例如，从小的整数类型转换为大的整数类型，从派生类转换为基类。
- 显式类型转换 - 这些转换是通过用户使用预定义的函数显式完成的。显式转换需要强制转换运算符。

下面的实例显示了一个显式的类型转换：

```
namespace TypeConversionApplication
{
    class ExplicitConversion
    {
        static void Main(string[] args)
        {
            double d = 5673.74;
            int i;

            // 强制转换 double 为 int
            i = (int)d;
            Console.WriteLine(i);
            Console.ReadKey();

        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
5673
```

# C# 类型转换方法

C# 提供了下列内置的类型转换方法：

| 方法 | 描述 |
| --- | --- |
| **ToBoolean** | 如果可能的话，把类型转换为布尔型。 |
| **ToByte** | 把类型转换为字节类型。 |
| **ToChar** | 如果可能的话，把类型转换为单个 Unicode 字符类型。 |
| **ToDateTime** | 把类型（整数或字符串类型）转换为 日期-时间 结构。 |
| **ToDecimal** | 把浮点型或整数类型转换为十进制类型。 |
| **ToDouble** | 把类型转换为双精度浮点型。 |
| **ToInt16** | 把类型转换为 16 位整数类型。 |
| **ToInt32** | 把类型转换为 32 位整数类型。 |
| **ToInt64** | 把类型转换为 64 位整数类型。 |
| **ToSbyte** | 把类型转换为有符号字节类型。 |
| **ToSingle** | 把类型转换为小浮点数类型。 |
| **ToString** | 把类型转换为字符串类型。 |
| **ToType** | 把类型转换为指定类型。 |
| **ToUInt16** | 把类型转换为 16 位无符号整数类型。 |
| **ToUInt32** | 把类型转换为 32 位无符号整数类型。 |
| **ToUInt64** | 把类型转换为 64 位无符号整数类型。 |

下面的实例把不同值的类型转换为字符串类型：

```
namespace TypeConversionApplication
{
    class StringConversion
    {
        static void Main(string[] args)
        {
            int i = 75;
            float f = 53.005f;
            double d = 2345.7652;
            bool b = true;

            Console.WriteLine(i.ToString());
            Console.WriteLine(f.ToString());
            Console.WriteLine(d.ToString());
            Console.WriteLine(b.ToString());
            Console.ReadKey();

        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
75
53.005
2345.7652
True
```

# C# 变量

一个变量只不过是一个供程序操作的存储区的名字。在 C# 中，每个变量都有一个特定的类型，类型决定了变量的内存大小和布局。范围内的值可以存储在内存中，可以对变量进行一系列操作。

我们已经讨论了各种数据类型。C# 中提供的基本的值类型大致可以分为以下几类：

| 类型 | 举例 |
| --- | --- |
| 整数类型 | sbyte、byte、short、ushort、int、uint、long、ulong 和 char |
| 浮点型 | float 和 double |
| 十进制类型 | decimal |
| 布尔类型 | true 或 false 值，指定的值 |
| 空类型 | 可为空值的数据类型 |

C# 允许定义其他值类型的变量，比如 **enum**，也允许定义引用类型变量，比如 **class**。这些我们将在以后的章节中进行讨论。在本章节中，我们只研究基本变量类型。

# C# 中的变量定义

C# 中变量定义的语法：

```
<data_type> <variable_list>;
```

在这里，data_type 必须是一个有效的 C# 数据类型，可以是 char、int、float、double 或其他用户自定义的数据类型。variable_list 可以由一个或多个用逗号分隔的标识符名称组成。

一些有效的变量定义如下所示：

```
int i, j, k;
char c, ch;
float f, salary;
double d;
```

您可以在变量定义时进行初始化：

```
int i = 100;
```

# C# 中的变量初始化

变量通过在等号后跟一个常量表达式进行初始化（赋值）。初始化的一般形式为：

```
variable_name = value;
```

变量可以在声明时被初始化（指定一个初始值）。初始化由一个等号后跟一个常量表达式组成，如下所示：

```
<data_type> <variable_name> = value;
```

一些实例：

```
int d = 3, f = 5;    /* 初始化 d 和 f. */
byte z = 22;         /* 初始化 z. */
double pi = 3.14159; /* 声明 pi 的近似值 */
char x = 'x';        /* 变量 x 的值为 'x' */
```

正确地初始化变量是一个良好的编程习惯，否则有时程序会产生意想不到的结果。

请看下面的实例，使用了各种类型的变量：

```
namespace VariableDefinition
{
    class Program
    {
        static void Main(string[] args)
        {
            short a;
            int b ;
            double c;

            /* 实际初始化 */
            a = 10;
            b = 20;
            c = a + b;
            Console.WriteLine("a = {0}, b = {1}, c = {2}", a, b, c);
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a = 10, b = 20, c = 30
```

# 接受来自用户的值

**System** 命名空间中的 **Console** 类提供了一个函数 **ReadLine()**，用于接收来自用户的输入，并把它存储到一个变量中。

例如：

```
int num;
num = Convert.ToInt32(Console.ReadLine());
```

函数 **Convert.ToInt32()** 把用户输入的数据转换为 int 数据类型，因为 **Console.ReadLine()** 只接受字符串格式的数据。

# C# 中的 Lvalues 和 Rvalues

C# 中的两种表达式：

1. **lvalue**：lvalue 表达式可以出现在赋值语句的左边或右边。
2. **rvalue**：rvalue 表达式可以出现在赋值语句的右边，不能出现在赋值语句的左边。

变量是 lvalue 的，所以可以出现在赋值语句的左边。数值是 rvalue 的，因此不能被赋值，不能出现在赋值语句的左边。下面是一个有效的语句：

```
int g = 20;
```

下面是一个无效的语句，会产生编译时错误：

```
10 = 20;
```

# C# 常量

常量是固定值，程序执行期间不会改变。常量可以是任何基本数据类型，比如整数常量、浮点常量、字符常量或者字符串常量，还有枚举常量。

常量可以被当作常规的变量，只是它们的值在定义后不能被修改。

# 整数常量

整数常量可以是十进制、八进制或十六进制的常量。前缀指定基数：0x 或 0X 表示十六进制，0 表示八进制，没有前缀则表示十进制。

整数常量也可以有后缀，可以是 U 和 L 的组合，其中，U 和 L 分别表示 unsigned 和 long。后缀可以是大写或者小写，多个后缀以任意顺序进行组合。

这里有一些整数常量的实例：

```
212         /* 合法 */
215u        /* 合法 */
0xFeeL      /* 合法 */
078         /* 非法：8 不是一个八进制数字 */
032UU       /* 非法：不能重复后缀 */
```

以下是各种类型的整数常量的实例：

```
85         /* 十进制 */
0213       /* 八进制 */
0x4b       /* 十六进制 */
30         /* int */
30u        /* 无符号 int */
30l        /* long */
30ul       /* 无符号 long */
```

# 浮点常量

一个浮点常量是由整数部分、小数点、小数部分和指数部分组成。您可以使用小数形式或者指数形式来表示浮点常量。

这里有一些浮点常量的实例：

```
3.14159       /* 合法 */
314159E-5L    /* 合法 */
510E          /* 非法：不完全指数 */
210f          /* 非法：没有小数或指数 */
.e55          /* 非法：缺少整数或小数 */
```

使用小数形式表示时，必须包含小数点、指数或同时包含两者。使用指数形式表示时，必须包含整数部分、小数部分或同时包含两者。有符号的指数是用 e 或 E 表示的。

# 字符常量

字符常量是括在单引号里，例如，'x'，且可存储在一个简单的字符类型变量中。一个字符常量可以是一个普通字符（例如 'x'）、一个转义序列（例如 '\t'）或者一个通用字符（例如 '\u02C0'）。

在 C# 中有一些特定的字符，当它们的前面带有反斜杠时有特殊的意义，可用于表示换行符（\n）或制表符 tab（\t）。在这里，列出一些转义序列码：

| 转义序列 | 含义 |
|---|---|
| \\ | \ 字符 |
| \' | ' 字符 |
| \" | " 字符 |
| \? | ? 字符 |
| \a | Alert 或 bell |
| \b | 退格键（Backspace） |
| \f | 换页符（Form feed） |
| \n | 换行符（Newline） |
| \r | 回车 |
| \t | 水平制表符 tab |
| \v | 垂直制表符 tab |
| \ooo | 一到三位的八进制数 |
| \xhh . . . | 一个或多个数字的十六进制数 |

以下是一些转义序列字符的实例：

```
namespace EscapeChar
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello\tWorld\n\n");
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Hello   World
```

# 字符串常量

字符常量是括在双引号 "" 里，或者是括在 @"" 里。字符串常量包含的字符与字符常量相似，可以是：普通字符、转义序列和通用字符

使用字符串常量时，可以把一个很长的行拆成多个行，可以使用空格分隔各个部分。

这里是一些字符串常量的实例。下面所列的各种形式表示相同的字符串。

```
"hello, dear"
"hello, \
dear"
"hello, " "d" "ear"
@"hello dear"
```

# 定义常量

常量是使用 **const** 关键字来定义的 。定义一个常量的语法如下：

```
const <data_type> <constant_name> = value;
```

下面的代码演示了如何在程序中定义和使用常量：

```
using System;

namespace DeclaringConstants
{
    class Program
    {
        static void Main(string[] args)
        {
            const double pi = 3.14159; // 常量声明
            double r;
            Console.WriteLine("Enter Radius: ");
            r = Convert.ToDouble(Console.ReadLine());
            double areaCircle = pi * r * r;
            Console.WriteLine("Radius: {0}, Area: {1}", r, areaCircle);
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Enter Radius:
3
Radius: 3, Area: 28.27431
```

# C# 运算符

运算符是一种告诉编译器执行特定的数学或逻辑操作的符号。C# 有丰富的内置运算符，分类如下：

- 算术运算符
- 关系运算符
- 逻辑运算符
- 位运算符
- 赋值运算符
- 杂项运算符

本教程将逐一讲解算术运算符、关系运算符、逻辑运算符、位运算符、赋值运算符及其他运算符。

## 算术运算符

下表显示了 C# 支持的所有算术运算符。假设变量 **A** 的值为 10，变量 **B** 的值为 20，则：

| 运算符 | 描述 | 实例 |
|---|---|---|
| + | 把两个操作数相加 | A + B 将得到 30 |
| - | 从第一个操作数中减去第二个操作数 | A - B 将得到 -10 |
| * | 把两个操作数相乘 | A * B 将得到 200 |
| / | 分子除以分母 | B / A 将得到 2 |
| % | 取模运算符，整除后的余数 | B % A 将得到 0 |
| ++ | 自增运算符，整数值增加 1 | A++ 将得到 11 |
| -- | 自减运算符，整数值减少 1 | A-- 将得到 9 |

### 实例

请看下面的实例，了解 C# 中所有可用的算术运算符：

```
using System;

namespace OperatorsAppl
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 21;
            int b = 10;
            int c;

            c = a + b;
            Console.WriteLine("Line 1 - c 的值是 {0}", c);
            c = a - b;
            Console.WriteLine("Line 2 - c 的值是 {0}", c);
            c = a * b;
            Console.WriteLine("Line 3 - c 的值是 {0}", c);
            c = a / b;
            Console.WriteLine("Line 4 - c 的值是 {0}", c);
            c = a % b;
            Console.WriteLine("Line 5 - c 的值是 {0}", c);
            c = a++;
            Console.WriteLine("Line 6 - c 的值是 {0}", c);
            c = a--;
            Console.WriteLine("Line 7 - c 的值是 {0}", c);
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - c 的值是 31
Line 2 - c 的值是 11
Line 3 - c 的值是 210
Line 4 - c 的值是 2
Line 5 - c 的值是 1
Line 6 - c 的值是 21
Line 7 - c 的值是 22
```

# 关系运算符

下表显示了 C# 支持的所有关系运算符。假设变量 **A** 的值为 10，变量 **B** 的值为 20，则：

| 运算符 | 描述 | 实例 |
|---|---|---|
| == | 检查两个操作数的值是否相等，如果相等则条件为真。 | (A == B) 不为真。 |
| != | 检查两个操作数的值是否相等，如果不相等则条件为真。 | (A != B) 为真。 |
| > | 检查左操作数的值是否大于右操作数的值，如果是则条件为真。 | (A > B) 不为真。 |
| < | 检查左操作数的值是否小于右操作数的值，如果是则条件为真。 | (A < B) 为真。 |
| >= | 检查左操作数的值是否大于或等于右操作数的值，如果是则条件为真。 | (A >= B) 不为真。 |
| <= | 检查左操作数的值是否小于或等于右操作数的值，如果是则条件为真。 | (A <= B) 为真。 |

## 实例

请看下面的实例，了解 C# 中所有可用的关系运算符：

```
using System;

class Program
{
  static void Main(string[] args)
  {
      int a = 21;
      int b = 10;

      if (a == b)
      {
          Console.WriteLine("Line 1 - a 等于 b");
      }
      else
      {
          Console.WriteLine("Line 1 - a 不等于 b");
      }
      if (a < b)
      {
          Console.WriteLine("Line 2 - a 小于 b");
      }
      else
      {
          Console.WriteLine("Line 2 - a 不小于 b");
      }
      if (a > b)
      {
          Console.WriteLine("Line 3 - a 大于 b");
      }
      else
      {
          Console.WriteLine("Line 3 - a 不大于 b");
      }
      /* 改变 a 和 b 的值 */
      a = 5;
      b = 20;
      if (a <= b)
      {
         Console.WriteLine("Line 4 - a 小于或等于 b");
      }
      if (b >= a)
      {
         Console.WriteLine("Line 5 - b 大于或等于 a");
      }
  }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - a 不等于 b
Line 2 - a 不小于 b
Line 3 - a 大于 b
Line 4 - a 小于或等于 b
Line 5 - b 大于或等于 a
```

# 逻辑运算符

下表显示了 C# 支持的所有逻辑运算符。假设变量 **A** 为布尔值 true，变量 **B** 为布尔值 false，则：

| 运算符 | 描述 | 实例 |
| --- | --- | --- |
| && | 称为逻辑与运算符。如果两个操作数都非零，则条件为真。 | (A && B) 为假。 |
| |:|: | 称为逻辑或运算符。如果两个操作数中有任意一个非零，则条件为真。 | (A |:|: B) 为真。 |
| ! | 称为逻辑非运算符。用来逆转操作数的逻辑状态。如果条件为真则逻辑非运算符将使其为假。 | !(A && B) 为真。 |

## 实例

请看下面的实例，了解 C# 中所有可用的逻辑运算符：

```
using System;

namespace OperatorsAppl
{
    class Program
    {
        static void Main(string[] args)
        {
            bool a = true;
            bool b = true;

            if (a && b)
            {
               Console.WriteLine("Line 1 - 条件为真");
            }
            if (a || b)
            {
                Console.WriteLine("Line 2 - 条件为真");
            }
            /* 改变 a 和 b 的值 */
            a = false;
            b = true;
            if (a && b)
            {
                Console.WriteLine("Line 3 - 条件为真");
            }
            else
            {
                Console.WriteLine("Line 3 - 条件不为真");
            }
            if (!(a && b))
            {
                Console.WriteLine("Line 4 - 条件为真");
            }
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - 条件为真
Line 2 - 条件为真
Line 3 - 条件不为真
Line 4 - 条件为真
```

# 位运算符

位运算符作用于位，并逐位执行操作。&、 | 和 ^ 的真值表如下所示：

| p | q | p & q | p \| q | p ^ q |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

假设如果 A = 60，且 B = 13，现在以二进制格式表示，它们如下所示：

```
A = 0011 1100

B = 0000 1101

-----------------

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A  = 1100 0011
```

下表列出了 C# 支持的位运算符。假设变量 **A** 的值为 60，变量 **B** 的值为 13，则：

| 运算符 | 描述 | 实例 |
|---|---|---|
| & | 如果同时存在于两个操作数中，二进制 AND 运算符复制一位到结果中。 | (A & B) 将得到 12，即为 0000 1100 |
| \| | 如果存在于任一操作数中，二进制 OR 运算符复制一位到结果中。 | (A \| B) 将得到 61，即为 0011 1101 |
| ^ | 如果存在于其中一个操作数中但不同时存在于两个操作数中，二进制异或运算符复制一位到结果中。 | (A ^ B) 将得到 49，即为 0011 0001 |
| ~ | 二进制补码运算符是一元运算符，具有"翻转"位效果。 | (~A ) 将得到 -61，即为 1100 0011，2 的补码形式，带符号的二进制数。 |
| << | 二进制左移运算符。左操作数的值向左移动右操作数指定的位数。 | A << 2 将得到 240，即为 1111 0000 |
| >> | 二进制右移运算符。左操作数的值向右移动右操作数指定的位数。 | A >> 2 将得到 15，即为 0000 1111 |

## 实例

请看下面的实例，了解 C# 中所有可用的位运算符：

```
using System;
namespace OperatorsAppl
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 60;               /* 60 = 0011 1100 */
            int b = 13;               /* 13 = 0000 1101 */
            int c = 0;

             c = a & b;           /* 12 = 0000 1100 */
             Console.WriteLine("Line 1 - c 的值是 {0}", c );

             c = a | b;           /* 61 = 0011 1101 */
             Console.WriteLine("Line 2 - c 的值是 {0}", c);

             c = a ^ b;           /* 49 = 0011 0001 */
             Console.WriteLine("Line 3 - c 的值是 {0}", c);

             c = ~a;               /*-61 = 1100 0011 */
             Console.WriteLine("Line 4 - c 的值是 {0}", c);

             c = a << 2;     /* 240 = 1111 0000 */
             Console.WriteLine("Line 5 - c 的值是 {0}", c);

             c = a >> 2;     /* 15 = 0000 1111 */
             Console.WriteLine("Line 6 - c 的值是 {0}", c);
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - c 的值是 12
Line 2 - c 的值是 61
Line 3 - c 的值是 49
Line 4 - c 的值是 -61
Line 5 - c 的值是 240
Line 6 - c 的值是 15
```

# 赋值运算符

下表列出了 C# 支持的赋值运算符：

| 运算符 | 描述 | 实例 |
| --- | --- | --- |
| = | 简单的赋值运算符，把右边操作数的值赋给左边操作数 | C = A + B 将把 A + B 的值赋给 C |
| += | 加且赋值运算符，把右边操作数加上左边操作数的结果赋值给左边操作数 | C += A 相当于 C = C + A |
| -= | 减且赋值运算符，把左边操作数减去右边操作数的结果赋值给左边操作数 | C -= A 相当于 C = C - A |
| *= | 乘且赋值运算符，把右边操作数乘以左边操作数的结果赋值给左边操作数 | C = A 相当于 C = C A |
| /= | 除且赋值运算符，把左边操作数除以右边操作数的结果赋值给左边操作数 | C /= A 相当于 C = C / A |
| %= | 求模且赋值运算符，求两个操作数的模赋值给左边操作数 | C %= A 相当于 C = C % A |
| <<= | 左移且赋值运算符 | C <<= 2 等同于 C = C << 2 |
| >>= | 右移且赋值运算符 | C >>= 2 等同于 C = C >> 2 |
| &= | 按位与且赋值运算符 | C &= 2 等同于 C = C & 2 |
| ^= | 按位异或且赋值运算符 | C ^= 2 等同于 C = C ^ 2 |
| |= | 按位或且赋值运算符 | C |= 2 等同于 C = C | 2 |

# 实例

请看下面的实例，了解 C# 中所有可用的赋值运算符：

```
using System;

namespace OperatorsAppl
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 21;
            int c;

            c = a;
            Console.WriteLine("Line 1 - =  c 的值 = {0}", c);

            c += a;
            Console.WriteLine("Line 2 - += c 的值 = {0}", c);

            c -= a;
            Console.WriteLine("Line 3 - -=  c 的值 = {0}", c);

            c *= a;
            Console.WriteLine("Line 4 - *=  c 的值 = {0}", c);

            c /= a;
            Console.WriteLine("Line 5 - /=  c 的值 = {0}", c);

            c = 200;
            c %= a;
            Console.WriteLine("Line 6 - %=  c 的值 = {0}", c);

            c <<= 2;
            Console.WriteLine("Line 7 - <<=  c 的值 = {0}", c);

            c >>= 2;
            Console.WriteLine("Line 8 - >>=  c 的值 = {0}", c);

            c &= 2;
            Console.WriteLine("Line 9 - &=  c 的值 = {0}", c);

            c ^= 2;
            Console.WriteLine("Line 10 - ^=  c 的值 = {0}", c);

            c |= 2;
            Console.WriteLine("Line 11 - |=  c 的值 = {0}", c);
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - =     c 的值 = 21
Line 2 - +=    c 的值 = 42
Line 3 - -=    c 的值 = 21
Line 4 - *=    c 的值 = 441
Line 5 - /=    c 的值 = 21
Line 6 - %=    c 的值 = 11
Line 7 - <<=    c 的值 = 44
Line 8 - >>=    c 的值 = 11
Line 9 - &=    c 的值 = 2
Line 10 - ^=    c 的值 = 0
Line 11 - |=    c 的值 = 2
```

## 杂项运算符

下表列出了 C# 支持的其他一些重要的运算符，包括 **sizeof**、**typeof** 和 **? :**。

| 运算符 | 描述 | 实例 |
| --- | --- | --- |
| sizeof() | 返回数据类型的大小。 | sizeof(int)，将返回 4. |
| typeof() | 返回 class 的类型。 | typeof(StreamReader); |
| & | 返回变量的地址。 | &a; 将得到变量的实际地址。 |
| * | 变量的指针。 | *a; 将指向一个变量。 |
| ? : | 条件表达式 | 如果条件为真 ? 则为 X : 否则为 Y |
| is | 判断对象是否为某一类型。 | If( Ford is Car) // 检查 Ford 是否是 Car 类的一个对象。 |
| as | 强制转换，即使转换失败也不会抛出异常。 | Object obj = new StringReader("Hello");<br>StringReader r = obj as StringReader; |

# 实例

```
using System;

namespace OperatorsAppl
{

   class Program
   {
      static void Main(string[] args)
      {

         /* sizeof 运算符的实例 */
         Console.WriteLine("int 的大小是 {0}", sizeof(int));
         Console.WriteLine("short 的大小是 {0}", sizeof(short));
         Console.WriteLine("double 的大小是 {0}", sizeof(double));

         /* 三元运算符符的实例 */
         int a, b;
         a = 10;
         b = (a == 1) ? 20 : 30;
         Console.WriteLine("b 的值是 {0}", b);

         b = (a == 10) ? 20 : 30;
         Console.WriteLine("b 的值是 {0}", b);
         Console.ReadLine();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
int 的大小是 4
short 的大小是 2
double 的大小是 8
b 的值是 30
b 的值是 20
```

# C# 中的运算符优先级

运算符的优先级确定表达式中项的组合。这会影响到一个表达式如何计算。某些运算符比其他运算符有更高的优先级，例如，乘除运算符具有比加减运算符更高的优先级。

例如 x = 7 + 3 2，在这里，*x 被赋值为 13，而不是 20*，因为运算符 具有比 + 更高的优先级，所以首先计算乘法 3*2，然后再加上 7。

下表将按运算符优先级从高到低列出各个运算符，具有较高优先级的运算符出现在表格的上面，具有较低优先级的运算符出现在表格的下面。在表达式中，较高优先级的运算符会优先被计算。

<table>
<tr><th>类别</th><th>运算符</th><th>结合性</th></tr>
<tr><td>后缀</td><td>() [] -> . ++ - -</td><td>从左到右</td></tr>
<tr><td>一元</td><td>+ - ! ~ ++ - - (type)* & sizeof</td><td>从右到左</td></tr>
<tr><td>乘除</td><td>* / %</td><td>从左到右</td></tr>
<tr><td>加减</td><td>+ -</td><td>从左到右</td></tr>
<tr><td>移位</td><td><< >></td><td>从左到右</td></tr>
<tr><td>关系</td><td>< <= > >=</td><td>从左到右</td></tr>
<tr><td>相等</td><td>== !=</td><td>从左到右</td></tr>
<tr><td>位与 AND</td><td>&</td><td>从左到右</td></tr>
<tr><td>位异或 XOR</td><td>^</td><td>从左到右</td></tr>
<tr><td>位或 OR</td><td>|</td><td>从左到右</td></tr>
<tr><td>逻辑与 AND</td><td>&&</td><td>从左到右</td></tr>
<tr><td>逻辑或 OR</td><td>||</td><td>从左到右</td></tr>
<tr><td>条件</td><td>?:</td><td>从右到左</td></tr>
<tr><td>赋值</td><td>= += -= *= /= %=>>= <<= &= ^= |=</td><td>从右到左</td></tr>
<tr><td>逗号</td><td>,</td><td>从左到右</td></tr>
</table>

## 实例

```
using System;

namespace OperatorsAppl
{

    class Program
    {
        static void Main(string[] args)
        {
            int a = 20;
            int b = 10;
            int c = 15;
            int d = 5;
            int e;
            e = (a + b) * c / d;     // ( 30 * 15 ) / 5
            Console.WriteLine("(a + b) * c / d 的值是 {0}", e);

            e = ((a + b) * c) / d;   // (30 * 15 ) / 5
            Console.WriteLine("((a + b) * c) / d 的值是 {0}", e);

            e = (a + b) * (c / d);   // (30) * (15/5)
            Console.WriteLine("(a + b) * (c / d) 的值是 {0}", e);

            e = a + (b * c) / d;    //  20 + (150/5)
            Console.WriteLine("a + (b * c) / d 的值是 {0}", e);
            Console.ReadLine();
        }
    }
}
```

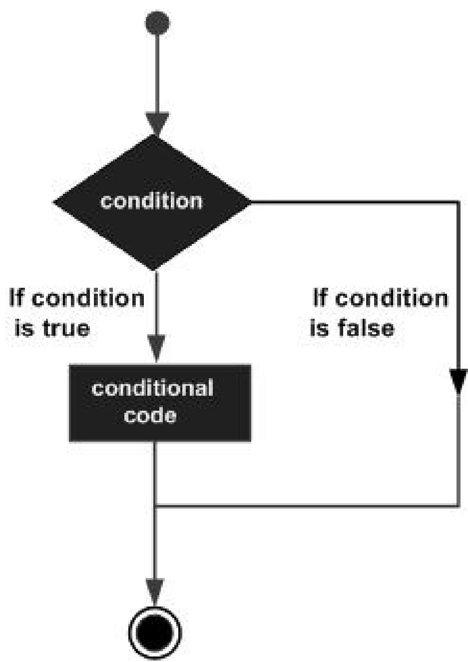
当上面的代码被编译和执行时，它会产生下列结果：

```
(a + b) * c / d 的值是 90
((a + b) * c) / d 的值是 90
(a + b) * (c / d) 的值是 90
a + (b * c) / d 的值是 50
```

# C# 判断

判断结构要求程序员指定一个或多个要评估或测试的条件，以及条件为真时要执行的语句（必需的）和条件为假时要执行的语句（可选的）。

下面是大多数编程语言中典型的判断结构的一般形式：




## 判断语句

C# 提供了以下类型的判断语句。点击链接查看每个语句的细节。

| 语句 | 描述 |
| --- | --- |
| if 语句 | 一个 **if 语句** 由一个布尔表达式后跟一个或多个语句组成。 |
| if...else 语句 | 一个 **if 语句** 后可跟一个可选的 **else 语句**，else 语句在布尔表达式为假时执行。 |
| 嵌套 if 语句 | 您可以在一个 **if** 或 **else if** 语句内使用另一个 **if** 或 **else if** 语句。 |
| switch 语句 | 一个 **switch** 语句允许测试一个变量等于多个值时的情况。 |
| 嵌套 switch 语句 | 您可以在一个 **switch** 语句内使用另一个 **switch** 语句。 |

## ? : 运算符

我们已经在前面的章节中讲解了 条件运算符 **? :**，可以用来替代 **if...else** 语句。它的一般形式如下：

```
Exp1 ? Exp2 : Exp3;
```

其中，Exp1、Exp2 和 Exp3 是表达式。请注意，冒号的使用和位置。

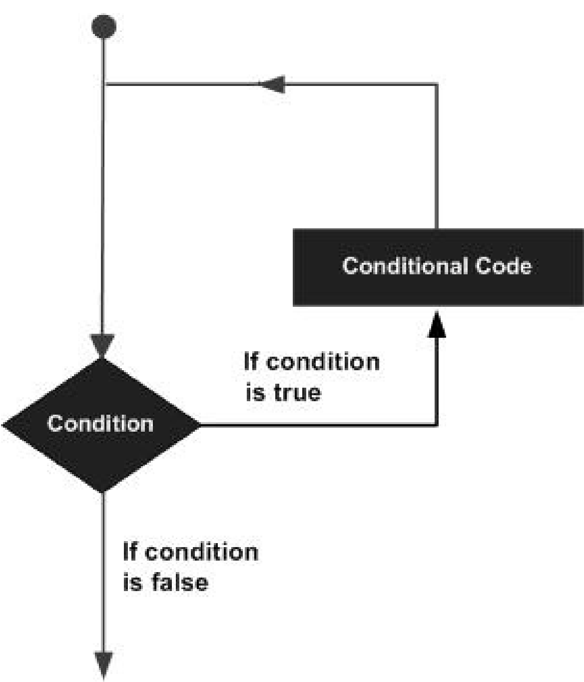
? 表达式的值是由 Exp1 决定的。如果 Exp1 为真，则计算 Exp2 的值，结果即为整个 ? 表达式的值。如果 Exp1 为假，则计算 Exp3 的值，结果即为整个 ? 表达式的值。

# C# 循环

有的时候，可能需要多次执行同一块代码。一般情况下，语句是顺序执行的：函数中的第一个语句先执行，接着是第二个语句，依此类推。

编程语言提供了允许更为复杂的执行路径的多种控制结构。

循环语句允许我们多次执行一个语句或语句组，下面是大多数编程语言中循环语句的一般形式：




## 循环类型

C# 提供了以下几种循环类型。点击链接查看每个类型的细节。

| 循环类型 | 描述 |
| --- | --- |
| while 循环 | 当给定条件为真时，重复语句或语句组。它会在执行循环主体之前测试条件。 |
| for 循环 | 多次执行一个语句序列，简化管理循环变量的代码。 |
| do...while 循环 | 除了它是在循环主体结尾测试条件外，其他与 while 语句类似。 |
| 嵌套循环 | 您可以在 while、for 或 do..while 循环内使用一个或多个循环。 |

## 循环控制语句

循环控制语句更改执行的正常序列。当执行离开一个范围时，所有在该范围中创建的自动对象都会被销毁。

C# 提供了下列的控制语句。点击链接查看每个语句的细节。

| 控制语句 | 描述 |
| --- | --- |
| break 语句 | 终止 **loop** 或 **switch** 语句，程序流将继续执行紧接着 loop 或 switch 的下一条语句。 |
| continue 语句 | 引起循环跳过主体的剩余部分，立即重新开始测试条件。 |

# 无限循环

如果条件永远不为假，则循环将变成无限循环。**for** 循环在传统意义上可用于实现无限循环。由于构成循环的三个表达式中任何一个都不是必需的，您可以将某些条件表达式留空来构成一个无限循环。

```
using System;

namespace Loops
{

    class Program
    {
        static void Main(string[] args)
        {
            for (; ; )
            {
                Console.WriteLine("Hey! I am Trapped");
            }

        }
    }
}
```

当条件表达式不存在时，它被假设为真。您也可以设置一个初始值和增量表达式，但是一般情况下，程序员偏向于使用 for(;;) 结构来表示一个无限循环。

# C# 封装

封装 被定义为"把一个或多个项目封闭在一个物理的或者逻辑的包中"。在面向对象程序设计方法论中，封装是为了防止对实现细节的访问。

抽象和封装是面向对象程序设计的相关特性。抽象允许相关信息可视化，封装则使程序员实现所需级别的抽象。

封装使用 访问修饰符 来实现。一个 访问修饰符 定义了一个类成员的范围和可见性。C# 支持的访问修饰符如下所示：

- Public
- Private
- Protected
- Internal
- Protected internal

## Public 访问修饰符

Public 访问修饰符允许一个类将其成员变量和成员函数暴露给其他的函数和对象。任何公有成员可以被外部的类访问。

下面的实例说明了这点：

```
using System;

namespace RectangleApplication
{
    class Rectangle
    {
        //成员变量
        public double length;
        public double width;

        public double GetArea()
        {
            return length * width;
        }
        public void Display()
        {
            Console.WriteLine("长度： {0}", length);
            Console.WriteLine("宽度： {0}", width);
            Console.WriteLine("面积： {0}", GetArea());
        }
    }//end class Rectangle
    class ExecuteRectangle
    {
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle();
            r.length = 4.5;
            r.width = 3.5;
            r.Display();
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
长度： 4.5
宽度： 3.5
面积： 15.75
```

在上面的实例中，成员变量 length 和 width 被声明为 **public**，所以它们可以被函数 Main() 使用 Rectangle 类的实例 **r** 访问。

成员函数 *Display()* 和 *GetArea()* 也可以不通过类的实例直接访问这些变量。

成员函数 *Display()* 也被声明为 **public**，所以它也能被 *Main()* 使用 Rectangle 类的实例 **r** 访问。

# Private 访问修饰符

Private 访问修饰符允许一个类将其成员变量和成员函数对其他的函数和对象进行隐藏。只有同一个类中的函数可以访问它的私有成员。即使是类的实例也不能访问它的私有成员。

下面的实例说明了这点：

```
using System;

namespace RectangleApplication
{
    class Rectangle
    {
        //成员变量
        private double length;
        private double width;

        public void Acceptdetails()
        {
            Console.WriteLine("请输入长度：");
            length = Convert.ToDouble(Console.ReadLine());
            Console.WriteLine("请输入宽度：");
            width = Convert.ToDouble(Console.ReadLine());
        }
        public double GetArea()
        {
            return length * width;
        }
        public void Display()
        {
            Console.WriteLine("长度： {0}", length);
            Console.WriteLine("宽度： {0}", width);
            Console.WriteLine("面积： {0}", GetArea());
        }
    }//end class Rectangle
    class ExecuteRectangle
    {
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle();
            r.Acceptdetails();
            r.Display();
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
请输入长度：
4.4
请输入宽度：
3.3
长度： 4.4
宽度： 3.3
面积： 14.52
```

在上面的实例中，成员变量 length 和 width 被声明为 **private**，所以它们不能被函数 Main() 访问。

成员函数 *AcceptDetails()* 和 *Display()* 可以访问这些变量。

由于成员函数 *AcceptDetails()* 和 *Display()* 被声明为 **public**，所以它们可以被 *Main()* 使用 Rectangle 类的实例 **r** 访问。

# Protected 访问修饰符

Protected 访问修饰符允许子类访问它的基类的成员变量和成员函数。这样有助于实现继承。我们将在继承的章节详细讨论这个。更详细地讨论这个。

# Internal 访问修饰符

Internal 访问说明符允许一个类将其成员变量和成员函数暴露给当前程序中的其他函数和对象。换句话说，带有 internal 访问修饰符的任何成员可以被定义在该成员所定义的应用程序内的任何类或方法访问。

下面的实例说明了这点：

```
using System;

namespace RectangleApplication
{
    class Rectangle
    {
        //成员变量
        internal double length;
        internal double width;

        double GetArea()
        {
            return length * width;
        }
       public void Display()
        {
            Console.WriteLine("长度： {0}", length);
            Console.WriteLine("宽度： {0}", width);
            Console.WriteLine("面积： {0}", GetArea());
        }
    }//end class Rectangle
    class ExecuteRectangle
    {
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle();
            r.length = 4.5;
            r.width = 3.5;
            r.Display();
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
长度： 4.5
宽度： 3.5
面积： 15.75
```

在上面的实例中，请注意成员函数 *GetArea()* 声明的时候不带有任何访问修饰符。如果没有指定访问修饰符，则使用类成员的默认访问修饰符，即为 **private**。

# Protected Internal 访问修饰符

Protected Internal 访问修饰符允许一个类将其成员变量和成员函数对同一应用程序内的子类以外的其他的类对象和函数进行隐藏。这也被用于实现继承。

# C# 方法

一个方法是把一些相关的语句组织在一起，用来执行一个任务的语句块。每一个 C# 程序至少有一个带有 Main 方法的类。

要使用一个方法，您需要：

- 定义方法
- 调用方法

# C# 中定义方法

当定义一个方法时，从根本上说是在声明它的结构的元素。在 C# 中，定义方法的语法如下：

```
<Access Specifier> <Return Type> <Method Name>(Parameter List)
{
   Method Body
}
```

下面是方法的各个元素：

- **Access Specifier**：访问修饰符，这个决定了变量或方法对于另一个类的可见性。
- **Return type**：返回类型，一个方法可以返回一个值。返回类型是方法返回的值的数据类型。如果方法不返回任何值，则返回类型为 **void**。
- **Method name**：方法名称，是一个唯一的标识符，且是大小写敏感的。它不能与类中声明的其他标识符相同。
- **Parameter list**：参数列表，使用圆括号括起来，该参数是用来传递和接收方法的数据。参数列表是指方法的参数类型、顺序和数量。参数是可选的，也就是说，一个方法可能不包含参数。
- **Method body**：方法主体，包含了完成任务所需的指令集。

## 实例

下面的代码片段显示一个函数 *FindMax*，它接受两个整数值，并返回两个中的较大值。它有 public 访问修饰符，所以它可以使用类的实例从类的外部进行访问。

```
class NumberManipulator
{
   public int FindMax(int num1, int num2)
   {
      /* 局部变量声明 */
      int result;

      if (num1 > num2)
         result = num1;
      else
         result = num2;

      return result;
   }
   ...
}
```

# C# 中调用方法

您可以使用方法名调用方法。下面的实例演示了这点：

```
using System;

namespace CalculatorApplication
{
   class NumberManipulator
   {
      public int FindMax(int num1, int num2)
      {
         /* 局部变量声明 */
         int result;

         if (num1 > num2)
            result = num1;
         else
            result = num2;

         return result;
      }
      static void Main(string[] args)
      {
         /* 局部变量定义 */
         int a = 100;
         int b = 200;
         int ret;
         NumberManipulator n = new NumberManipulator();

         //调用 FindMax 方法
         ret = n.FindMax(a, b);
         Console.WriteLine("最大值是： {0}", ret );
         Console.ReadLine();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
最大值是： 200
```

您也可以使用类的实例从另一个类中调用其他类的公有方法。例如，方法 *FindMax* 属于 *NumberManipulator* 类，您可以从另一个类 *Test* 中调用它。

```
using System;

namespace CalculatorApplication
{
    class NumberManipulator
    {
        public int FindMax(int num1, int num2)
        {
            /* 局部变量声明 */
            int result;

            if (num1 > num2)
                result = num1;
            else
                result = num2;

            return result;
        }
    }
    class Test
    {
        static void Main(string[] args)
        {
            /* 局部变量定义 */
            int a = 100;
            int b = 200;
            int ret;
            NumberManipulator n = new NumberManipulator();
            //调用 FindMax 方法
            ret = n.FindMax(a, b);
            Console.WriteLine("最大值是： {0}", ret );
            Console.ReadLine();

        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
最大值是： 200
```

# 递归方法调用

一个方法可以自我调用。这就是所谓的 递归。下面的实例使用递归函数计算一个数的阶乘：

```
using System;

namespace CalculatorApplication
{
    class NumberManipulator
    {
        public int factorial(int num)
        {
            /* 局部变量定义 */
            int result;

            if (num == 1)
            {
                return 1;
            }
            else
            {
                result = factorial(num - 1) * num;
                return result;
            }
        }

        static void Main(string[] args)
        {
            NumberManipulator n = new NumberManipulator();
            //调用 factorial 方法
            Console.WriteLine("6 的阶乘是： {0}", n.factorial(6));
            Console.WriteLine("7 的阶乘是： {0}", n.factorial(7));
            Console.WriteLine("8 的阶乘是： {0}", n.factorial(8));
            Console.ReadLine();

        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
6 的阶乘是： 720
7 的阶乘是： 5040
8 的阶乘是： 40320
```

# 参数传递

当调用带有参数的方法时，您需要向方法传递参数。在 C# 中，有三种向方法传递参数的方式：

| 方式 | 描述 |
| --- | --- |
| 值参数 | 这种方式复制参数的实际值给函数的形式参数，实参和形参使用的是两个不同内存中的值。在这种情况下，当形参的值发生改变时，不会影响实参的值，从而保证了实参数据的安全。 |
| 引用参数 | 这种方式复制参数的内存位置的引用给形式参数。这意味着，当形参的值发生改变时，同时也改变实参的值。 |
| 输出参数 | 这种方式可以返回多个值。 |

# 按值传递参数

这是参数传递的默认方式。在这种方式下，当调用一个方法时，会为每个值参数创建一个新的存储位置。

实际参数的值会复制给形参，实参和形参使用的是两个不同内存中的值。所以，当形参的值发生改变时，不会影响实参的值，从而保证了实参数据的安全。下面的实例演示了这个概念：

```
using System;

namespace CalculatorApplication
{
    class NumberManipulator
    {
        public void swap(int x, int y)
        {
            int temp;

            temp = x; /* 保存 x 的值 */
            x = y;    /* 把 y 赋值给 x */
            y = temp; /* 把 temp 赋值给 y */
        }

        static void Main(string[] args)
        {
            NumberManipulator n = new NumberManipulator();
            /* 局部变量定义 */
            int a = 100;
            int b = 200;

            Console.WriteLine("在交换之前，a 的值： {0}", a);
            Console.WriteLine("在交换之前，b 的值： {0}", b);

            /* 调用函数来交换值 */
            n.swap(a, b);

            Console.WriteLine("在交换之后，a 的值： {0}", a);
            Console.WriteLine("在交换之后，b 的值： {0}", b);

            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
在交换之前，a 的值：100
在交换之前，b 的值：200
在交换之后，a 的值：100
在交换之后，b 的值：200
```

结果表明，即使在函数内改变了值，值也没有发生任何的变化。

# 按引用传递参数

引用参数是一个对变量的内存位置的引用。当按引用传递参数时，与值参数不同的是，它不会为这些参数创建一个新的存储位置。引用参数表示与提供给方法的实际参数具有相同的内存位置。

在 C# 中，使用 **ref** 关键字声明引用参数。下面的实例演示了这点：

```
using System;
namespace CalculatorApplication
{
   class NumberManipulator
   {
      public void swap(ref int x, ref int y)
      {
         int temp;

         temp = x; /* 保存 x 的值 */
         x = y;    /* 把 y 赋值给 x */
         y = temp; /* 把 temp 赋值给 y */
       }

      static void Main(string[] args)
      {
         NumberManipulator n = new NumberManipulator();
         /* 局部变量定义 */
         int a = 100;
         int b = 200;

         Console.WriteLine("在交换之前，a 的值： {0}", a);
         Console.WriteLine("在交换之前，b 的值： {0}", b);

         /* 调用函数来交换值 */
         n.swap(ref a, ref b);

         Console.WriteLine("在交换之后，a 的值： {0}", a);
         Console.WriteLine("在交换之后，b 的值： {0}", b);

         Console.ReadLine();

      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
在交换之前，a 的值：100
在交换之前，b 的值：200
在交换之后，a 的值：200
在交换之后，b 的值：100
```

结果表明，*swap* 函数内的值改变了，且这个改变可以在 *Main* 函数中反映出来。

# 按输出传递参数

return 语句可用于只从函数中返回一个值。但是，可以使用 输出参数 来从函数中返回两个值。输出参数会把方法输出的数据赋给自己，其他方面与引用参数相似。

下面的实例演示了这点：

```
using System;

namespace CalculatorApplication
{
   class NumberManipulator
   {
      public void getValue(out int x )
      {
         int temp = 5;
         x = temp;
      }

      static void Main(string[] args)
      {
         NumberManipulator n = new NumberManipulator();
         /* 局部变量定义 */
         int a = 100;

         Console.WriteLine("在方法调用之前，a 的值： {0}", a);

         /* 调用函数来获取值 */
         n.getValue(out a);

         Console.WriteLine("在方法调用之后，a 的值： {0}", a);
         Console.ReadLine();

      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
在方法调用之前，a 的值： 100
在方法调用之后，a 的值： 5
```

提供给输出参数的变量不需要赋值。当需要从一个参数没有指定初始值的方法中返回值时，输出参数特别有用。请看下面的实例，来理解这一点：

```
using System;

namespace CalculatorApplication
{
   class NumberManipulator
   {
      public void getValues(out int x, out int y )
      {
          Console.WriteLine("请输入第一个值： ");
          x = Convert.ToInt32(Console.ReadLine());
          Console.WriteLine("请输入第二个值： ");
          y = Convert.ToInt32(Console.ReadLine());
      }

      static void Main(string[] args)
      {
         NumberManipulator n = new NumberManipulator();
         /* 局部变量定义 */
         int a , b;

         /* 调用函数来获取值 */
         n.getValues(out a, out b);

         Console.WriteLine("在方法调用之后，a 的值： {0}", a);
         Console.WriteLine("在方法调用之后，b 的值： {0}", b);
         Console.ReadLine();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果（取决于用户输入）：

```
请输入第一个值：
7
请输入第二个值：
8
在方法调用之后，a 的值： 7
在方法调用之后，b 的值： 8
```

# C# 可空类型（Nullable）

## C# 可空类型（Nullable）

C# 提供了一个特殊的数据类型，**nullable** 类型（可空类型），可空类型可以表示其基础值类型正常范围内的值，再加上一个 null 值。

例如，Nullable< Int32 >，读作"可空的 Int32"，可以被赋值为 -2,147,483,648 到 2,147,483,647 之间的任意值，也可以被赋值为 null 值。类似的，Nullable< bool > 变量可以被赋值为 true 或 false 或 null。

在处理数据库和其他包含可能未赋值的元素的数据类型时，将 null 赋值给数值类型或布尔型的功能特别有用。例如，数据库中的布尔型字段可以存储值 true 或 false，或者，该字段也可以未定义。

声明一个 **nullable** 类型（可空类型）的语法如下：

```
< data_type> ? <variable_name> = null;
```

下面的实例演示了可空数据类型的用法：

```
using System;
namespace CalculatorApplication
{
   class NullablesAtShow
   {
      static void Main(string[] args)
      {
         int? num1 = null;
         int? num2 = 45;
         double? num3 = new double?();
         double? num4 = 3.14157;

         bool? boolval = new bool?();

         // 显示值

         Console.WriteLine("显示可空类型的值： {0}, {1}, {2}, {3}",
                            num1, num2, num3, num4);
         Console.WriteLine("一个可空的布尔值： {0}", boolval);
         Console.ReadLine();

      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
显示可空类型的值： , 45,  , 3.14157
一个可空的布尔值：
```

# Null 合并运算符（ ?? ）

Null 合并运算符用于定义可空类型和引用类型的默认值。Null 合并运算符为类型转换定义了一个预设值，以防可空类型的值为 Null。Null 合并运算符把操作数类型隐式转换为另一个可空（或不可空）的值类型的操作数的类型。

如果第一个操作数的值为 null，则运算符返回第二个操作数的值，否则返回第一个操作数的值。下面的实例演示了这点：

```
using System;
namespace CalculatorApplication
{
   class NullablesAtShow
   {

      static void Main(string[] args)
      {

         double? num1 = null;
         double? num2 = 3.14157;
         double num3;
         num3 = num1 ?? 5.34;
         Console.WriteLine("num3 的值： {0}", num3);
         num3 = num2 ?? 5.34;
         Console.WriteLine("num3 的值： {0}", num3);
         Console.ReadLine();

      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

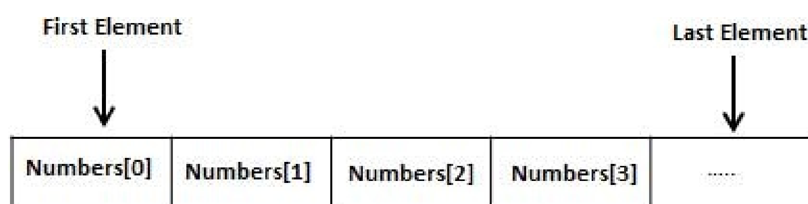
```
num3 的值： 5.34
num3 的值： 3.14157
```

# C# 数组（Array）

数组是一个存储相同类型元素的固定大小的顺序集合。数组是用来存储数据的集合，通常认为数组是一个同一类型变量的集合。

声明数组变量并不是声明 number0、number1、...、number99 一个个单独的变量，而是声明一个就像 numbers 这样的变量，然后使用 numbers[0]、numbers[1]、...、numbers[99] 来表示一个个单独的变量。数组中某个指定的元素是通过索引来访问的。

所有的数组都是由连续的内存位置组成的。最低的地址对应第一个元素，最高的地址对应最后一个元素。




## 声明数组

在 C# 中声明一个数组，您可以使用下面的语法：

```
datatype[] arrayName;
```

其中，

- *datatype* 用于指定被存储在数组中的元素的类型。
- *[ ]* 指定数组的秩（维度）。秩指定数组的大小。
- *arrayName* 指定数组的名称。

例如：

```
double[] balance;
```

## 初始化数组

声明一个数组不会在内存中初始化数组。当初始化数组变量时，您可以赋值给数组。

数组是一个引用类型，所以您需要使用 **new** 关键字来创建数组的实例。

例如：

```
double[] balance = new double[10];
```

# 赋值给数组

您可以通过使用索引号赋值给一个单独的数组元素，比如：

```
double[] balance = new double[10];
balance[0] = 4500.0;
```

您可以在声明数组的同时给数组赋值，比如：

```
double[] balance = { 2340.0, 4523.69, 3421.0};
```

您也可以创建并初始化一个数组，比如：

```
int [] marks = new int[5]  { 99,  98, 92, 97, 95};
```

在上述情况下，你也可以省略数组的大小，比如：

```
int [] marks = new int[]  { 99,  98, 92, 97, 95};
```

您也可以赋值一个数组变量到另一个目标数组变量中。在这种情况下，目标和源会指向相同的内存位置：

```
int [] marks = new int[]  { 99,  98, 92, 97, 95};
int[] score = marks;
```

当您创建一个数组时，C# 编译器会根据数组类型隐式初始化每个数组元素为一个默认值。例如，int 数组的所有元素都会被初始化为 0。

# 访问数组元素

元素是通过带索引的数组名称来访问的。这是通过把元素的索引放置在数组名称后的方括号中来实现的。例如：

```
double salary = balance[9];
```

下面是一个实例，使用上面提到的三个概念，即声明、赋值、访问数组：

```
using System;
namespace ArrayApplication
{
   class MyArray
   {
      static void Main(string[] args)
      {
         int []  n = new int[10]; /* n 是一个带有 10 个整数的数组 */
         int i,j;

         /* 初始化数组 n 中的元素 */
         for ( i = 0; i < 10; i++ )
         {
            n[ i ] = i + 100;
         }

         /* 输出每个数组元素的值 */
         for (j = 0; j < 10; j++ )
         {
            Console.WriteLine("Element[{0}] = {1}", j, n[j]);
         }
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

# 使用 *foreach* 循环

在前面的实例中，我们使用一个 for 循环来访问每个数组元素。您也可以使用一个 **foreach** 语句来遍历数组。

```
using System;

namespace ArrayApplication
{
   class MyArray
   {
      static void Main(string[] args)
      {
         int []  n = new int[10]; /* n 是一个带有 10 个整数的数组 */

         /* 初始化数组 n 中的元素 */
         for ( int i = 0; i < 10; i++ )
         {
            n[i] = i + 100;
         }

         /* 输出每个数组元素的值 */
         foreach (int j in n )
         {
            int i = j-100;
            Console.WriteLine("Element[{0}] = {1}", i, j);
            i++;
         }
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

# C# 数组细节

在 C# 中，数组是非常重要的，且需要了解更多的细节。下面列出了 C# 程序员必须清楚的一些与数组相关的重要概念：

| 概念 | 描述 |
|---|---|
| 多维数组 | C# 支持多维数组。多维数组最简单的形式是二维数组。 |
| 交错数组 | C# 支持交错数组，即数组的数组。 |
| 传递数组给函数 | 您可以通过指定不带索引的数组名称来给函数传递一个指向数组的指针。 |
| 参数数组 | 这通常用于传递未知数量的参数给函数。 |
| Array 类 | 在 System 命名空间中定义，是所有数组的基类，并提供了各种用于数组的属性和方法。 |

# C# 字符串（String）

在 C# 中，您可以使用字符数组来表示字符串，但是，更常见的做法是使用 **string** 关键字来声明一个字符串变量。string 关键字是 **System.String** 类的别名。

## 创建 String 对象

您可以使用以下方法之一来穿件 string 对象：

- 通过给 String 变量指定一个字符串
- 通过使用 String 类构造函数
- 通过使用字符串串联运算符（ + ）
- 通过检索属性或调用一个返回字符串的方法
- 通过格式化方法来转换一个值或对象为它的字符串表示形式

下面的实例演示了这点：

```
using System;

namespace StringApplication
{
    class Program
    {
        static void Main(string[] args)
        {
           //字符串，字符串连接
            string fname, lname;
            fname = "Rowan";
            lname = "Atkinson";

            string fullname = fname + lname;
            Console.WriteLine("Full Name: {0}", fullname);

            //通过使用 string 构造函数
            char[] letters = { 'H', 'e', 'l', 'l','o' };
            string greetings = new string(letters);
            Console.WriteLine("Greetings: {0}", greetings);

            //方法返回字符串
            string[] sarray = { "Hello", "From", "Tutorials", "Point" };
            string message = String.Join(" ", sarray);
            Console.WriteLine("Message: {0}", message);

            //用于转化值的格式化方法
            DateTime waiting = new DateTime(2012, 10, 10, 17, 58, 1);
            string chat = String.Format("Message sent at {0:t} on {0:D}",
            waiting);
            Console.WriteLine("Message: {0}", chat);
            Console.ReadKey() ;
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Full Name: Rowan Atkinson
Greetings: Hello
Message: Hello From Tutorials Point
Message: Message sent at 5:58 PM on Wednesday, October 10, 2012
```

# String 类的属性

String 类有以下两个属性：

| 名称 | 描述 |
|---|---|
| **Chars** | 在当前 *String* 对象中获取 *Char* 对象的指定位置。 |
| **Length** | 在当前的 *String* 对象中获取字符数。 |

# String 类的方法

String 类有许多方法用于 string 对象的操作。下面的表格提供了一些最常用的方法：

| 名称 | 描述 |
|---|---|
| **public static int Compare( string strA, string strB )** | 比较两个指定的 string 对象，并返回一个表示它们在排列顺序中相对位置的整数。该方法区分大小写。 |
| **public static int Compare( string strA, string strB, bool ignoreCase )** | 比较两个指定的 string 对象，并返回一个表示它们在排列顺序中相对位置的整数。但是，如果布尔参数为真时，该方法不区分大小写。 |
| **public static string Concat( string str0, string str1 )** | 连接两个 string 对象。 |
| **public static string Concat( string str0, string str1, string str2 )** | 连接三个 string 对象。 |
| **public static string Concat( string str0, string str1, string str2, string str3 )** | 连接四个 string 对象。 |
| **public bool Contains( string value )** | 返回一个表示指定 string 对象是否出现在字符串中的值。 |
| **public static string Copy( string str )** | 创建一个与指定字符串具有相同值的新的 String 对象。 |
| **public void CopyTo( int sourceIndex, char[] destination, int destinationIndex, int count )** | 从 string 对象的指定位置开始复制指定数量的字符到 Unicode 字符数组中的指定位置。 |
| **public bool EndsWith(** | 判断 string 对象的结尾是否匹配指定的字符串。 |

| | |
|---|---|
| **string value )** | 判断 string 对象的结尾是否匹配指定的字符串。 |
| **public bool Equals( string value )** | 判断当前的 string 对象是否与指定的 string 对象具有相同的值。 |
| **public static bool Equals( string a, string b )** | 判断两个指定的 string 对象是否具有相同的值。 |
| **public static string Format( string format, Object arg0 )** | 把指定字符串中一个或多个格式项替换为指定对象的字符串表示形式。 |
| **public int IndexOf( char value )** | 返回指定 Unicode 字符在当前字符串中第一次出现的索引，索引从 0 开始。 |
| **public int IndexOf( string value )** | 返回指定字符串在该实例中第一次出现的索引，索引从 0 开始。 |
| **public int IndexOf( char value, int startIndex )** | 返回指定 Unicode 字符从该字符串中指定字符位置开始搜索第一次出现的索引，索引从 0 开始。 |
| **public int IndexOf( string value, int startIndex )** | 返回指定字符串从该实例中指定字符位置开始搜索第一次出现的索引，索引从 0 开始。 |
| **public int IndexOfAny( char[] anyOf )** | 返回某一个指定的 Unicode 字符数组中任意字符在该实例中第一次出现的索引，索引从 0 开始。 |
| **public int IndexOfAny( char[] anyOf, int startIndex )** | 返回某一个指定的 Unicode 字符数组中任意字符从该实例中指定字符位置开始搜索第一次出现的索引，索引从 0 开始。 |
| **public string Insert( int startIndex, string value )** | 返回一个新的字符串，其中，指定的字符串被插入在当前 string 对象的指定索引位置。 |
| **public static bool IsNullOrEmpty( string value )** | 指示指定的字符串是否为 null 或者是否为一个空的字符串。 |
| **public static string Join( string separator, params string[] value )** | 连接一个字符串数组中的所有元素，使用指定的分隔符分隔每个元素。 |
| **public static string Join( string separator, string[] value, int startIndex, int count )** | 链接一个字符串数组中的指定元素，使用指定的分隔符分隔每个元素。 |
| **public int LastIndexOf( char value )** | 返回指定 Unicode 字符在当前 string 对象中最后一次出现的索引位置，索引从 0 开始。 |
| **public int LastIndexOf( string value )** | 返回指定字符串在当前 string 对象中最后一次出现的索引位置，索引从 0 开始。 |
| **public string Remove( int startIndex )** | 移除当前实例中的所有字符，从指定位置开始，一直到最后一个位置为止，并返回字符串。 |
| **public string Remove( int startIndex, int count )** | 从当前字符串的指定位置开始移除指定数量的字符，并返回字符串。 |

| | |
|---|---|
| **oldChar, char newChar )** | 另一个指定的 Unicode 字符，并返回新的字符串。 |
| **public string Replace( string oldValue, string newValue )** | 把当前 string 对象中，所有指定的字符串替换为另一个指定的字符串，并返回新的字符串。 |
| **public string[] Split( params char[] separator )** | 返回一个字符串数组，包含当前的 string 对象中的子字符串，子字符串是使用指定的 Unicode 字符数组中的元素进行分隔的。 |
| **public string[] Split( char[] separator, int count )** | 返回一个字符串数组，包含当前的 string 对象中的子字符串，子字符串是使用指定的 Unicode 字符数组中的元素进行分隔的。int 参数指定要返回的子字符串的最大数目。 |
| **public bool StartsWith( string value )** | 判断字符串实例的开头是否匹配指定的字符串。 |
| **public char[] ToCharArray()** | 返回一个带有当前 string 对象中所有字符的 Unicode 字符数组。 |
| **public char[] ToCharArray( int startIndex, int length )** | 返回一个带有当前 string 对象中所有字符的 Unicode 字符数组，从指定的索引开始，直到指定的长度为止。 |
| **public string ToLower()** | 把字符串转换为小写并返回。 |
| **public string ToUpper()** | 把字符串转换为大写并返回。 |
| **public string Trim()** | 移除当前 String 对象中的所有前导空白字符和后置空白字符。 |

上面的方法列表并不详尽，请访问 MSDN 库，查看完整的方法列表和 String 类构造函数。

# 实例

下面的实例演示了上面提到的一些方法：

比较字符串

```
using System;

namespace StringApplication
{
   class StringProg
   {
      static void Main(string[] args)
      {
         string str1 = "This is test";
         string str2 = "This is text";

         if (String.Compare(str1, str2) == 0)
         {
            Console.WriteLine(str1 + " and " + str2 +  " are equal.");
         }
         else
         {
            Console.WriteLine(str1 + " and " + str2 + " are not equal.");
         }
         Console.ReadKey() ;
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
This is test and This is text are not equal.
```

字符串包含字符串：

```
using System;

namespace StringApplication
{
   class StringProg
   {
      static void Main(string[] args)
      {
         string str = "This is test";
         if (str.Contains("test"))
         {
            Console.WriteLine("The sequence 'test' was found.");
         }
         Console.ReadKey() ;
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
The sequence 'test' was found.
```

获取子字符串：

```
using System;

namespace StringApplication
{
   class StringProg
   {
      static void Main(string[] args)
      {
         string str = "Last night I dreamt of San Pedro";
         Console.WriteLine(str);
         string substr = str.Substring(23);
         Console.WriteLine(substr);
      }
      Console.ReadKey() ;
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
San Pedro
```

连接字符串：

```
using System;

namespace StringApplication
{
   class StringProg
   {
      static void Main(string[] args)
      {
         string[] starray = new string[]{"Down the way nights are dark",
         "And the sun shines daily on the mountain top",
         "I took a trip on a sailing ship",
         "And when I reached Jamaica",
         "I made a stop"};

         string str = String.Join("\n", starray);
         Console.WriteLine(str);
      }
      Console.ReadKey() ;
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Down the way nights are dark
And the sun shines daily on the mountain top
I took a trip on a sailing ship
And when I reached Jamaica
I made a stop
```

# C# 结构（Struct）

在 C# 中，结构是值类型数据结构。它使得一个单一变量可以存储各种数据类型的相关数据。**struct** 关键字用于创建结构。

结构是用来代表一个记录。假设您想跟踪图书馆中书的动态。您可能想跟踪每本书的以下属性：

- Title
- Author
- Subject
- Book ID

# 定义结构

为了定义一个结构，您必须使用 struct 语句。struct 语句为程序定义了一个带有多个成员的新的数据类型。

例如，您可以按照如下的方式声明 Book 结构：

```
struct Books
{
   public string title;
   public string author;
   public string subject;
   public int book_id;
};
```

下面的程序演示了结构的用法：

```
using System;

struct Books
{
   public string title;
   public string author;
   public string subject;
   public int book_id;
};

public class testStructure
{
   public static void Main(string[] args)
   {

      Books Book1;        /* 声明 Book1，类型为 Book */
      Books Book2;        /* 声明 Book2，类型为 Book */

      /* book 1 详述 */
      Book1.title = "C Programming";
      Book1.author = "Nuha Ali";
      Book1.subject = "C Programming Tutorial";
      Book1.book_id = 6495407;

      /* book 2 详述 */
      Book2.title = "Telecom Billing";
      Book2.author = "Zara Ali";
      Book2.subject =  "Telecom Billing Tutorial";
      Book2.book_id = 6495700;

      /* 打印 Book1 信息 */
      Console.WriteLine( "Book 1 title : {0}", Book1.title);
      Console.WriteLine("Book 1 author : {0}", Book1.author);
      Console.WriteLine("Book 1 subject : {0}", Book1.subject);
      Console.WriteLine("Book 1 book_id :{0}", Book1.book_id);

      /* 打印 Book2 信息 */
      Console.WriteLine("Book 2 title : {0}", Book2.title);
      Console.WriteLine("Book 2 author : {0}", Book2.author);
      Console.WriteLine("Book 2 subject : {0}", Book2.subject);
      Console.WriteLine("Book 2 book_id : {0}", Book2.book_id);

      Console.ReadKey();

   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

# C# 结构的特点

您已经用了一个简单的名为 Books 的结构。在 C# 中的结构与传统的 C 或 C++ 中的结构不同。C# 中的结构有一下特点：

- 结构可带有方法、字段、索引、属性、运算符方法和事件。
- 结构可定义构造函数，但不能定义析构函数。但是，您不能为结构定义默认的构造函数。默认的构造函数是自动定义的，且不能被改变。
- 与类不同，结构不能继承其他的结构或类。
- 结构不能作为其他结构或类的基础结构。
- 结构可实现一个或多个接口。
- 结构成员不能指定为 abstract、virtual 或 protected。
- 当您使用 **New** 操作符创建一个结构对象时，会调用适当的构造函数来创建结构。与类不同，结构可以不使用 New 操作符即可被实例化。
- 如果不使用 New 操作符，只有在所有的字段都被初始化之后，字段才被赋值，对象才被使用。

# 类 vs 结构

类和结构有以下几个基本的不同点：

- 类是引用类型，结构是值类型。
- 结构不支持继承。
- 结构不能声明默认的构造函数。

针对上述讨论，让我们重写前面的实例：

```
using System;

struct Books
{
   private string title;
   private string author;
   private string subject;
   private int book_id;
   public void getValues(string t, string a, string s, int id)
   {
      title = t;
      author = a;
      subject = s;
      book_id = id;
   }
   public void display()
   {
      Console.WriteLine("Title : {0}", title);
      Console.WriteLine("Author : {0}", author);
      Console.WriteLine("Subject : {0}", subject);
      Console.WriteLine("Book_id :{0}", book_id);
   }

};

public class testStructure
{
   public static void Main(string[] args)
   {

      Books Book1 = new Books(); /* 声明 Book1，类型为 Book */
      Books Book2 = new Books(); /* 声明 Book2，类型为 Book */

      /* book 1 详述 */
      Book1.getValues("C Programming",
      "Nuha Ali", "C Programming Tutorial",6495407);

      /* book 2 详述 */
      Book2.getValues("Telecom Billing",
      "Zara Ali", "Telecom Billing Tutorial", 6495700);

      /* 打印 Book1 信息 */
      Book1.display();

      /* 打印 Book2 信息 */
      Book2.display();

      Console.ReadKey();

   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Title : C Programming
Author : Nuha Ali
Subject : C Programming Tutorial
Book_id : 6495407
Title : Telecom Billing
Author : Zara Ali
Subject : Telecom Billing Tutorial
Book_id : 6495700
```

# C# 枚举（Enum）

枚举是一组命名整型常量。枚举类型是使用 **enum** 关键字声明的。

C# 枚举是值数据类型。换句话说，枚举包含自己的值，且不能继承或传递继承。

## 声明 *enum* 变量

声明枚举的一般语法：

```
enum <enum_name>
{
    enumeration list
};
```

其中，

- *enum_name* 指定枚举的类型名称。
- *enumeration list* 是一个用逗号分隔的标识符列表。

枚举列表中的每个符号代表一个整数值，一个比它前面的符号大的整数值。默认情况下，第一个枚举符号的值是 0.例如：

```
enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };
```

## 实例

下面的实例演示了枚举变量的用法：

```
using System;
namespace EnumApplication
{
   class EnumProgram
   {
      enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };

      static void Main(string[] args)
      {
         int WeekdayStart = (int)Days.Mon;
         int WeekdayEnd = (int)Days.Fri;
         Console.WriteLine("Monday: {0}", WeekdayStart);
         Console.WriteLine("Friday: {0}", WeekdayEnd);
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Monday: 1
Friday: 5
```

# C# 类（Class）

当您定义一个类时，您定义了一个数据类型的蓝图。这实际上并没有定义任何的数据，但它定义了类的名称意味着什么，也就是说，类的对象由什么组成及在这个对象上可执行什么操作。对象是类的实例。构成类的方法和变量成为类的成员。

## 类的定义

类的定义是以关键字 **class** 开始，后跟类的名称。类的主体，包含在一对花括号内。下面是类定义的一般形式：

```
<access specifier> class  class_name
{
    // member variables
    <access specifier> <data type> variable1;
    <access specifier> <data type> variable2;
    ...
    <access specifier> <data type> variableN;
    // member methods
    <access specifier> <return type> method1(parameter_list)
    {
        // method body
    }
    <access specifier> <return type> method2(parameter_list)
    {
        // method body
    }
    ...
    <access specifier> <return type> methodN(parameter_list)
    {
        // method body
    }
}
```

请注意：

- 访问标识符 <access specifier> 指定了对类及其成员的访问规则。如果没有指定，则使用默认的访问标识符。类的默认访问标识符是 **internal**，成员的默认访问标识符是 **private**。
- 数据类型 <data type> 指定了变量的类型，返回类型 <return type> 指定了返回的方法返回的数据类型。
- 如果要访问类的成员，您要使用点（.）运算符。
- 点运算符链接了对象的名称和成员的名称。

下面的实例说明了目前为止所讨论的概念：

```
using System;
namespace BoxApplication
{
    class Box
    {
       public double length;   // 长度
       public double breadth;  // 宽度
       public double height;   // 高度
    }
    class Boxtester
    {
        static void Main(string[] args)
        {
            Box Box1 = new Box();        // 声明 Box1，类型为 Box
            Box Box2 = new Box();        // 声明 Box2，类型为 Box
            double volume = 0.0;         // 体积

            // Box1 详述
            Box1.height = 5.0;
            Box1.length = 6.0;
            Box1.breadth = 7.0;

            // Box2 详述
            Box2.height = 10.0;
            Box2.length = 12.0;
            Box2.breadth = 13.0;

            // Box1 的体积
            volume = Box1.height * Box1.length * Box1.breadth;
            Console.WriteLine("Box1 的体积： {0}",  volume);

            // Box2 的体积
            volume = Box2.height * Box2.length * Box2.breadth;
            Console.WriteLine("Box2 的体积： {0}", volume);
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Box1 的体积： 210
Box2 的体积： 1560
```

# 成员函数和封装

类的成员函数是一个在类定义中有它的定义或原型的函数，就像其他变量一样。作为类的一个成为，它能在类的任何对象上操作，且能访问该对象的类的所有成员。

成员变量是对象的属性（从设计角度），且它们保持私有来实现封装。这些变量只能使用公共成员函数来访问。

让我们使用上面的概念来设置和获取一个类中不同的类成员的值：

```
using System;
namespace BoxApplication
{
    class Box
    {
       private double length;   // 长度
       private double breadth;  // 宽度
       private double height;   // 高度
       public void setLength( double len )
       {
            length = len;
       }

       public void setBreadth( double bre )
       {
            breadth = bre;
       }

       public void setHeight( double hei )
       {
            height = hei;
       }
       public double getVolume()
       {
           return length * breadth * height;
       }
    }
    class Boxtester
    {
        static void Main(string[] args)
        {
            Box Box1 = new Box();        // 声明 Box1，类型为 Box
            Box Box2 = new Box();         // 声明 Box2，类型为 Box
            double volume;                 // 体积

            // Box1 详述
            Box1.setLength(6.0);
            Box1.setBreadth(7.0);
            Box1.setHeight(5.0);

            // Box2 详述
            Box2.setLength(12.0);
            Box2.setBreadth(13.0);
            Box2.setHeight(10.0);

            // Box1 的体积
            volume = Box1.getVolume();
            Console.WriteLine("Box1 的体积： {0}" ,volume);

            // Box2 的体积
            volume = Box2.getVolume();
            Console.WriteLine("Box2 的体积： {0}", volume);

            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Box1 的体积： 210
Box2 的体积： 1560
```

# C# 中的构造函数

类的 构造函数 是类的一个特殊的成员函数，当创建类的新对象时执行。

构造函数的名称与类的名称完全相同，它没有任何返回类型。

下面的实例说明了构造函数的概念：

```
using System;
namespace LineApplication
{
   class Line
   {
      private double length;   // 线条的长度
      public Line()
      {
         Console.WriteLine("对象已创建");
      }

      public void setLength( double len )
      {
         length = len;
      }
      public double getLength()
      {
         return length;
      }

      static void Main(string[] args)
      {
         Line line = new Line();
         // 设置线条长度
         line.setLength(6.0);
         Console.WriteLine("线条的长度： {0}", line.getLength());
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
对象已创建
线条的长度： 6
```

默认的构造函数没有任何参数。但是如果您需要一个带有参数的构造函数可以有参数，这种构造函数叫做参数化构造函数。这种技术可以帮助您在创建对象的同时给对象赋初始值，具体请看下面实例：

```
using System;
namespace LineApplication
{
   class Line
   {
      private double length;   // 线条的长度
      public Line(double len)  // 参数化构造函数
      {
         Console.WriteLine("对象已创建，length = {0}", len);
         length = len;
      }

      public void setLength( double len )
      {
         length = len;
      }
      public double getLength()
      {
         return length;
      }

      static void Main(string[] args)
      {
         Line line = new Line(10.0);
         Console.WriteLine("线条的长度： {0}", line.getLength());
         // 设置线条长度
         line.setLength(6.0);
         Console.WriteLine("线条的长度： {0}", line.getLength());
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
对象已创建，length = 10
线条的长度： 10
线条的长度： 6
```

# C# 中的析构函数

类的 析构函数 是类的一个特殊的成员函数，当类的对象超出范围时执行。

析构函数的名称是在类的名称前加上一个波浪形（~）作为前缀，它不返回值，也不带任何参数。

析构函数用于在结束程序（比如关闭文件、释放内存等）之前释放资源。析构函数不能继承或重载。

下面的实例说明了析构函数的概念：

```
using System;
namespace LineApplication
{
   class Line
   {
      private double length;   // 线条的长度
      public Line()  // 构造函数
      {
         Console.WriteLine("对象已创建");
      }
      ~Line() //析构函数
      {
         Console.WriteLine("对象已删除");
      }

      public void setLength( double len )
      {
         length = len;
      }
      public double getLength()
      {
         return length;
      }

      static void Main(string[] args)
      {
         Line line = new Line();
         // 设置线条长度
         line.setLength(6.0);
         Console.WriteLine("线条的长度： {0}", line.getLength());
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
对象已创建
线条的长度： 6
对象已删除
```

# C# 类的静态成员

我们可以使用 **static** 关键字把类成员定义为静态的。当我们声明一个类成员为静态时，意味着无论有多少个类的对象被创建，只会有一个该静态成员的副本。

关键字 **static** 意味着类中只有一个该成员的实例。静态变量用于定义常量，因为它们的值可以通过直接调用类而不需要创建类的实例来获取。静态变量可在成员函数或类的定义外部进行初始化。您也可以在类的定义内部初始化静态变量。

下面的实例演示了静态变量的用法：

```
using System;
namespace StaticVarApplication
{
    class StaticVar
    {
       public static int num;
        public void count()
        {
            num++;
        }
        public int getNum()
        {
            return num;
        }
    }
    class StaticTester
    {
        static void Main(string[] args)
        {
            StaticVar s1 = new StaticVar();
            StaticVar s2 = new StaticVar();
            s1.count();
            s1.count();
            s1.count();
            s2.count();
            s2.count();
            s2.count();
            Console.WriteLine("s1 的变量 num： {0}", s1.getNum());
            Console.WriteLine("s2 的变量 num： {0}", s2.getNum());
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
s1 的变量 num： 6
s2 的变量 num： 6
```

您也可以把一个成员函数声明为 **static**。这样的函数只能访问静态变量。静态函数在对象被创建之前就已经存在。下面的实例演示了静态函数的用法：

```
using System;
namespace StaticVarApplication
{
    class StaticVar
    {
       public static int num;
        public void count()
        {
            num++;
        }
        public static int getNum()
        {
            return num;
        }
    }
    class StaticTester
    {
        static void Main(string[] args)
        {
            StaticVar s = new StaticVar();
            s.count();
            s.count();
            s.count();
            Console.WriteLine("变量 num： {0}", StaticVar.getNum());
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
变量 num： 3
```

# C# 继承

继承是面向对象程序设计中最重要的概念之一。继承允许我们根据一个类来定义另一个类来定义一个类，这使得创建和维护应用程序变得更容易。同时也有利于重用代码和节省开发时间。

当创建一个类时，程序员不需要完全重新编写新的数据成员和成员函数，只需要设计一个新的类，继承了已有的类的成员即可。这个已有的类被称为的基类，这个新的类被称为派生类。

继承的思想实现了 属于（**IS-A**） 关系。例如，哺乳动物 属于（**IS-A**） 动物，狗 属于（**IS-A**） 哺乳动物，因此狗 属于（**IS-A**） 动物。

# 基类和派生类

一个类可以派生自多个类或接口，这意味着它可以从多个基类或接口继承数据和函数。

C# 中创建派生类的语法如下：

```
<acess-specifier> class <base_class>
{
 ...
}
class <derived_class> : <base_class>
{
 ...
}
```

假设，有一个基类 Shape，它的派生类是 Rectangle：

```
using System;
namespace InheritanceApplication
{
   class Shape
   {
      public void setWidth(int w)
      {
         width = w;
      }
      public void setHeight(int h)
      {
         height = h;
      }
      protected int width;
      protected int height;
   }

   // 派生类
   class Rectangle: Shape
   {
      public int getArea()
      {
         return (width * height);
      }
   }

   class RectangleTester
   {
      static void Main(string[] args)
      {
         Rectangle Rect = new Rectangle();

         Rect.setWidth(5);
         Rect.setHeight(7);

         // 打印对象的面积
         Console.WriteLine("总面积： {0}",  Rect.getArea());
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
总面积： 35
```

# 基类的初始化

派生类继承了基类的成员变量和成员方法。因此父类对象应在子类对象创建之前被创建。您可以在成员初始化列表中进行父类的初始化。

下面的程序演示了这点：

```
using System;
namespace RectangleApplication
{
   class Rectangle
   {
      // 成员变量
      protected double length;
      protected double width;
      public Rectangle(double l, double w)
      {
         length = l;
         width = w;
      }
      public double GetArea()
      {
         return length * width;
      }
      public void Display()
      {
         Console.WriteLine("长度： {0}", length);
         Console.WriteLine("宽度： {0}", width);
         Console.WriteLine("面积： {0}", GetArea());
      }
   }//end class Rectangle
   class Tabletop : Rectangle
   {
      private double cost;
      public Tabletop(double l, double w) : base(l, w)
      { }
      public double GetCost()
      {
         double cost;
         cost = GetArea() * 70;
         return cost;
      }
      public void Display()
      {
         base.Display();
         Console.WriteLine("成本： {0}", GetCost());
      }
   }
   class ExecuteRectangle
   {
      static void Main(string[] args)
      {
         Tabletop t = new Tabletop(4.5, 7.5);
         t.Display();
         Console.ReadLine();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
长度： 4.5
宽度： 7.5
面积： 33.75
成本： 2362.5
```

# C# 多重继承

**C#** 不支持多重继承。但是，您可以使用接口来实现多重继承。下面的程序演示了这点：

```
using System;
namespace InheritanceApplication
{
   class Shape
   {
      public void setWidth(int w)
      {
         width = w;
      }
      public void setHeight(int h)
      {
         height = h;
      }
      protected int width;
      protected int height;
   }

   // 基类 PaintCost
   public interface PaintCost
   {
      int getCost(int area);

   }
   // 派生类
   class Rectangle : Shape, PaintCost
   {
      public int getArea()
      {
         return (width * height);
      }
      public int getCost(int area)
      {
         return area * 70;
      }
   }
   class RectangleTester
   {
      static void Main(string[] args)
      {
         Rectangle Rect = new Rectangle();
         int area;
         Rect.setWidth(5);
         Rect.setHeight(7);
         area = Rect.getArea();
         // 打印对象的面积
         Console.WriteLine("总面积： {0}",  Rect.getArea());
         Console.WriteLine("油漆总成本： ${0}" , Rect.getCost(area));
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
总面积： 35
油漆总成本： $2450
```

# C# 多态性

多态性意味着有多重形式。在面向对象编程范式中，多态性往往表现为"一个接口，多个功能"。

多态性可以是静态的或动态的。在静态多态性中，函数的响应是在编译时发生的。在动态多态性中，函数的响应是在运行时发生的。

## 静态多态性

在编译时，函数和对象的连接机制被称为早期绑定，也被称为静态绑定。C# 提供了两种技术来实现静态多态性。分别为：

- 函数重载
- 运算符重载

运算符重载将在下一章节讨论，接下来我们将讨论函数重载。

## 函数重载

您可以在同一个范围内对相同的函数名有多个定义。函数的定义必须彼此不同，可以是参数列表中的参数类型不同，也可以是参数个数不同。不能重载只有返回类型不同的函数声明。

下面的实例演示了几个相同的函数 **print()**，用于打印不同的数据类型：

```
using System;
namespace PolymorphismApplication
{
   class Printdata
   {
      void print(int i)
      {
         Console.WriteLine("Printing int: {0}", i );
      }

      void print(double f)
      {
         Console.WriteLine("Printing float: {0}" , f);
      }

      void print(string s)
      {
         Console.WriteLine("Printing string: {0}", s);
      }
      static void Main(string[] args)
      {
         Printdata p = new Printdata();
         // 调用 print 来打印整数
         p.print(5);
         // 调用 print 来打印浮点数
         p.print(500.263);
         // 调用 print 来打印字符串
         p.print("Hello C++");
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Printing int: 5
Printing float: 500.263
Printing string: Hello C++
```

# 动态多态性

C# 允许您使用关键字 **abstract** 创建抽象类，用于提供接口的部分类的实现。当一个派生类继承自该抽象类时，实现即完成。抽象类包含抽象方法，抽象方法可被派生类实现。派生类具有更专业的功能。

请注意，下面是有关抽象类的一些规则：

- 您不能创建一个抽象类的实例。
- 您不能在一个抽象类外部声明一个抽象方法。
- 通过在类定义前面放置关键字 **sealed**，可以将类声明为密封类。当一个类被声明为 **sealed** 时，它不能被继承。抽象类不能被声明为 sealed。

下面的程序演示了一个抽象类：

```
using System;
namespace PolymorphismApplication
{
   abstract class Shape
   {
      public abstract int area();
   }
   class Rectangle:  Shape
   {
      private int length;
      private int width;
      public Rectangle( int a=0, int b=0)
      {
         length = a;
         width = b;
      }
      public override int area ()
      {
         Console.WriteLine("Rectangle 类的面积：");
         return (width * length);
      }
   }

   class RectangleTester
   {
      static void Main(string[] args)
      {
         Rectangle r = new Rectangle(10, 7);
         double a = r.area();
         Console.WriteLine("面积： {0}",a);
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Rectangle 类的面积：
面积： 70
```

当有一个定义在类中的函数需要在继承类中实现时，可以使用虚方法。虚方法是使用关键字 **virtual** 声明的。虚方法可以在不同的继承类中有不同的实现。对虚方法的调用是在运行时发生的。

动态多态性是通过 抽象类 和 虚方法 实现的。

下面的程序演示了这点：

```
using System;
namespace PolymorphismApplication
{
   class Shape
   {
      protected int width, height;
      public Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      public virtual int area()
      {
         Console.WriteLine("父类的面积：");
         return 0;
      }
   }
   class Rectangle: Shape
   {
      public Rectangle( int a=0, int b=0): base(a, b)
      {

      }
      public override int area ()
      {
         Console.WriteLine("Rectangle 类的面积：");
         return (width * height);
      }
   }
   class Triangle: Shape
   {
      public Triangle(int a = 0, int b = 0): base(a, b)
      {

      }
      public override int area()
      {
         Console.WriteLine("Triangle 类的面积：");
         return (width * height / 2);
      }
   }
   class Caller
   {
      public void CallArea(Shape sh)
      {
         int a;
         a = sh.area();
         Console.WriteLine("面积： {0}", a);
      }
   }
   class Tester
   {

      static void Main(string[] args)
      {
         Caller c = new Caller();
         Rectangle r = new Rectangle(10, 7);
         Triangle t = new Triangle(10, 5);
         c.CallArea(r);
         c.CallArea(t);
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Rectangle 类的面积：
面积：70
Triangle 类的面积：
面积：25
```

# C# 运算符重载

您可以重定义或重载 C# 中内置的运算符。因此，程序员也可以使用用户自定义类型的运算符。重载运算符是具有特殊名称的函数，是通过关键字 **operator** 后跟运算符的符号来定义的。与其他函数一样，重载运算符有返回类型和参数列表。

例如，请看下面的函数：

```
public static Box operator+ (Box b, Box c)
{
   Box box = new Box();
   box.length = b.length + c.length;
   box.breadth = b.breadth + c.breadth;
   box.height = b.height + c.height;
   return box;
}
```

上面的函数为用户自定义的类 Box 实现了加法运算符（+）。它把两个 Box 对象的属性相加，并返回相加后的 Box 对象。

## 运算符重载的实现

下面的程序演示了完整的实现：

```
using System;

namespace OperatorOvlApplication
{
   class Box
   {
      private double length;      // 长度
      private double breadth;     // 宽度
      private double height;      // 高度

      public double getVolume()
      {
         return length * breadth * height;
      }
      public void setLength( double len )
      {
         length = len;
      }

      public void setBreadth( double bre )
      {
         breadth = bre;
      }

      public void setHeight( double hei )
      {
         height = hei;
      }
      // 重载 + 运算符来把两个 Box 对象相加
      public static Box operator+ (Box b, Box c)
      {
         Box box = new Box();
```

```
            box.length = b.length + c.length;
            box.breadth = b.breadth + c.breadth;
            box.height = b.height + c.height;
            return box;
        }

    }

    class Tester
    {
        static void Main(string[] args)
        {
            Box Box1 = new Box();         // 声明 Box1，类型为 Box
            Box Box2 = new Box();         // 声明 Box2，类型为 Box
            Box Box3 = new Box();         // 声明 Box3，类型为 Box
            double volume = 0.0;          // 体积

            // Box1 详述
            Box1.setLength(6.0);
            Box1.setBreadth(7.0);
            Box1.setHeight(5.0);

            // Box2 详述
            Box2.setLength(12.0);
            Box2.setBreadth(13.0);
            Box2.setHeight(10.0);

            // Box1 的体积
            volume = Box1.getVolume();
            Console.WriteLine("Box1 的体积： {0}", volume);

            // Box2 的体积
            volume = Box2.getVolume();
            Console.WriteLine("Box2 的体积： {0}", volume);

            // 把两个对象相加
            Box3 = Box1 + Box2;

            // Box3 的体积
            volume = Box3.getVolume();
            Console.WriteLine("Box3 的体积： {0}", volume);
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Box1 的体积： 210
Box2 的体积： 1560
Box3 的体积： 5400
```

# 可重载和不可重载运算符

下表描述了 C# 中运算符重载的能力：

| 运算符 | 描述 |
|---|---|
| +, -, !, ~, ++, -- | 这些一元运算符只有一个操作数，且可以被重载。 |
| +, -, *, /, % | 这些二元运算符带有两个操作数，且可以被重载。 |
| ==, !=, <, >, <=, >= | 这些比较运算符可以被重载。 |
| &&, \|\| | 这些条件逻辑运算符不能被直接重载。 |
| +=, -=, *=, /=, %= | 这些赋值运算符不能被重载。 |
| =, ., ?:, ->, new, is, sizeof, typeof | 这些运算符不能被重载。 |

# 实例

针对上述讨论，让我们扩展上面的实例，重载更多的运算符：

```
using System;

namespace OperatorOvlApplication
{
    class Box
    {
       private double length;      // 长度
       private double breadth;     // 宽度
       private double height;      // 高度

       public double getVolume()
       {
         return length * breadth * height;
       }
      public void setLength( double len )
      {
          length = len;
      }

      public void setBreadth( double bre )
      {
          breadth = bre;
      }

      public void setHeight( double hei )
      {
          height = hei;
      }
      // 重载 + 运算符来把两个 Box 对象相加
      public static Box operator+ (Box b, Box c)
      {
          Box box = new Box();
          box.length = b.length + c.length;
          box.breadth = b.breadth + c.breadth;
          box.height = b.height + c.height;
          return box;
      }

      public static bool operator == (Box lhs, Box rhs)
      {
          bool status = false;
          if (lhs.length == rhs.length && lhs.height == rhs.height
             && lhs.breadth == rhs.breadth)
          {
              status = true;
          }
```

```
            return status;
        }
        public static bool operator !=(Box lhs, Box rhs)
        {
            bool status = false;
            if (lhs.length != rhs.length || lhs.height != rhs.height
                || lhs.breadth != rhs.breadth)
            {
                status = true;
            }
            return status;
        }
        public static bool operator <(Box lhs, Box rhs)
        {
            bool status = false;
            if (lhs.length < rhs.length && lhs.height
                < rhs.height && lhs.breadth < rhs.breadth)
            {
                status = true;
            }
            return status;
        }

        public static bool operator >(Box lhs, Box rhs)
        {
            bool status = false;
            if (lhs.length > rhs.length && lhs.height
                > rhs.height && lhs.breadth > rhs.breadth)
            {
                status = true;
            }
            return status;
        }

        public static bool operator <=(Box lhs, Box rhs)
        {
            bool status = false;
            if (lhs.length <= rhs.length && lhs.height
                <= rhs.height && lhs.breadth <= rhs.breadth)
            {
                status = true;
            }
            return status;
        }

        public static bool operator >=(Box lhs, Box rhs)
        {
            bool status = false;
            if (lhs.length >= rhs.length && lhs.height
               >= rhs.height && lhs.breadth >= rhs.breadth)
            {
                status = true;
            }
            return status;
        }
        public override string ToString()
        {
            return String.Format("({0}, {1}, {2})", length, breadth, height);
        }

    }

    class Tester
    {
        static void Main(string[] args)
        {
          Box Box1 = new Box();          // 声明 Box1，类型为 Box
          Box Box2 = new Box();          // 声明 Box2，类型为 Box
          Box Box3 = new Box();          // 声明 Box3，类型为 Box
          Box Box4 = new Box();
          double volume = 0.0;   // 体积
```

```
            // Box1 详述
            Box1.setLength(6.0);
            Box1.setBreadth(7.0);
            Box1.setHeight(5.0);

            // Box2 详述
            Box2.setLength(12.0);
            Box2.setBreadth(13.0);
            Box2.setHeight(10.0);

           // 使用重载的 ToString() 显示两个盒子
            Console.WriteLine("Box1： {0}", Box1.ToString());
            Console.WriteLine("Box2： {0}", Box2.ToString());

            // Box1 的体积
            volume = Box1.getVolume();
            Console.WriteLine("Box1 的体积： {0}", volume);

            // Box2 的体积
            volume = Box2.getVolume();
            Console.WriteLine("Box2 的体积： {0}", volume);

            // 把两个对象相加
            Box3 = Box1 + Box2;
            Console.WriteLine("Box3： {0}", Box3.ToString());
            // Box3 的体积
            volume = Box3.getVolume();
            Console.WriteLine("Box3 的体积： {0}", volume);

            //comparing the boxes
            if (Box1 > Box2)
              Console.WriteLine("Box1 大于 Box2");
            else
              Console.WriteLine("Box1 不大于 Box2");
            if (Box1 < Box2)
              Console.WriteLine("Box1 小于 Box2");
            else
              Console.WriteLine("Box1 不小于 Box2");
            if (Box1 >= Box2)
              Console.WriteLine("Box1 大于等于 Box2");
            else
              Console.WriteLine("Box1 不大于等于 Box2");
            if (Box1 <= Box2)
              Console.WriteLine("Box1 小于等于 Box2");
            else
              Console.WriteLine("Box1 不小于等于 Box2");
            if (Box1 != Box2)
              Console.WriteLine("Box1 不等于 Box2");
            else
              Console.WriteLine("Box1 等于 Box2");
            Box4 = Box3;
            if (Box3 == Box4)
              Console.WriteLine("Box3 等于 Box4");
            else
              Console.WriteLine("Box3 不等于 Box4");

            Console.ReadKey();
          }
      }
  }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Box1： (6, 7, 5)
Box2： (12, 13, 10)
Box1 的体积： 210
Box2 的体积： 1560
Box3： (18, 20, 15)
Box3 的体积： 5400
Box1 不大于 Box2
Box1 小于 Box2
Box1 不大于等于 Box2
Box1 小于等于 Box2
Box1 不等于 Box2
Box3 等于 Box4
```

# C# 接口（Interface）

接口定义了所有类继承接口时应遵循的语法合同。接口定义了语法合同 **"是什么"** 部分，派生类定义了语法合同 **"怎么做"** 部分。

接口定义了属性、方法和事件，这些都是接口的成员。接口只包含了成员的声明。成员的定义是派生类的责任。接口提供了派生类应遵循的标准结构。

抽象类在某种程度上与接口类似，但是，它们大多只是用在当只有少数方法由基类声明由派生类实现时。

## 声明接口

接口使用 **interface** 关键字声明，它与类的声明类似。接口声明默认是 public 的。下面是一个接口声明的实例：

```
public interface ITransactions
{
   // 接口成员
   void showTransaction();
   double getAmount();
}
```

## 实例

下面的实例演示了上面接口的实现：

```
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace InterfaceApplication
{

   public interface ITransactions
   {
      // 接口成员
      void showTransaction();
      double getAmount();
   }
   public class Transaction : ITransactions
   {
      private string tCode;
      private string date;
      private double amount;
      public Transaction()
      {
         tCode = " ";
         date = " ";
         amount = 0.0;
      }
      public Transaction(string c, string d, double a)
      {
         tCode = c;
         date = d;
         amount = a;
      }
      public double getAmount()
      {
         return amount;
      }
      public void showTransaction()
      {
         Console.WriteLine("Transaction: {0}", tCode);
         Console.WriteLine("Date: {0}", date);
         Console.WriteLine("Amount: {0}", getAmount());

      }

   }
   class Tester
   {
      static void Main(string[] args)
      {
         Transaction t1 = new Transaction("001", "8/10/2012", 78900.00);
         Transaction t2 = new Transaction("002", "9/10/2012", 451900.00);
         t1.showTransaction();
         t2.showTransaction();
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Transaction: 001
Date: 8/10/2012
Amount: 78900
Transaction: 002
Date: 9/10/2012
Amount: 451900
```

# C# 命名空间（Namespace）

命名空间的设计目的是为了提供一种让一组名称与其他名称分隔开的方式。在一个命名空间中声明的类的名称与另一个命名空间中声明的相同的类的名称不冲突。

## 定义命名空间

命名空间的定义是以关键字 **namespace** 开始，后跟命名空间的名称，如下所示：

```
namespace namespace_name
{
   // 代码声明
}
```

为了调用支持命名空间版本的函数或变量，会把命名空间的名称置于前面，如下所示：

```
namespace_name.item_name;
```

下面的程序演示了命名空间的用法：

```
using System;
namespace first_space
{
   class namespace_cl
   {
      public void func()
      {
         Console.WriteLine("Inside first_space");
      }
   }
}
namespace second_space
{
   class namespace_cl
   {
      public void func()
      {
         Console.WriteLine("Inside second_space");
      }
   }
}
class TestClass
{
   static void Main(string[] args)
   {
      first_space.namespace_cl fc = new first_space.namespace_cl();
      second_space.namespace_cl sc = new second_space.namespace_cl();
      fc.func();
      sc.func();
      Console.ReadKey();
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Inside first_space
Inside second_space
```

# *using* 关键字

**using** 关键字表明程序使用的是给定命名空间中的名称。例如，我们在程序中使用 **System** 命名空间，其中定义了类 Console。我们可以只写：

```
Console.WriteLine ("Hello there");
```

我们可以写完全限定名称，如下：

```
System.Console.WriteLine("Hello there");
```

您也可以使用 **using** 命名空间指令，这样在使用的时候就不用在前面加上命名空间名称。该指令告诉编译器随后的代码使用了指定命名空间中的名称。下面的代码延时了命名空间的应用。

让我们使用 using 指定重写上面的实例：

```
using System;
using first_space;
using second_space;

namespace first_space
{
   class abc
   {
      public void func()
      {
         Console.WriteLine("Inside first_space");
      }
   }
}
namespace second_space
{
   class efg
   {
      public void func()
      {
         Console.WriteLine("Inside second_space");
      }
   }
}
class TestClass
{
   static void Main(string[] args)
   {
      abc fc = new abc();
      efg sc = new efg();
      fc.func();
      sc.func();
      Console.ReadKey();
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Inside first_space
Inside second_space
```

# 嵌套命名空间

命名空间可以被嵌套，即您可以在一个命名空间内定义另一个命名空间，如下所示：

```
namespace namespace_name1
{
   // 代码声明
   namespace namespace_name2
   {
     // 代码声明
   }
}
```

您可以使用点（.）运算符访问嵌套的命名空间的成员，如下所示：

```
using System;
using first_space;
using first_space.second_space;

namespace first_space
{
   class abc
   {
      public void func()
      {
         Console.WriteLine("Inside first_space");
      }
   }
   namespace second_space
   {
      class efg
      {
         public void func()
         {
            Console.WriteLine("Inside second_space");
         }
      }
   }
}

class TestClass
{
   static void Main(string[] args)
   {
      abc fc = new abc();
      efg sc = new efg();
      fc.func();
      sc.func();
      Console.ReadKey();
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Inside first_space
Inside second_space
```

# C# 预处理器指令

预处理器指令指导编译器在实际编译开始之前对信息进行预处理。

所有的预处理器指令都是以 # 开始。且在一行上，只有空白字符可以出现在预处理器指令之前。预处理器指令不是语句，所以它们不以分号（;）结束。

C# 编译器没有一个单独的预处理器，但是，指令被处理时就像是有一个单独的预处理器一样。在 C# 中，预处理器指令用于在条件编译中起作用。与 C 和 C++ 不同指令不用，它们不是用来创建宏。一个预处理器指令必须是该行上的唯一指令。

## C# 预处理器指令列表

下表列出了 C# 中可用的预处理器指令：

| 预处理器指令 | 描述 |
|---|---|
| #define | 它用于定义一系列成为符号的字符。 |
| #undef | 它用于取消定义符号。 |
| #if | 它用于测试符号是否为真。 |
| #else | 它用于创建复合条件指令，与 #if 一起使用。 |
| #elif | 它用于创建复合条件指令。 |
| #endif | 指定一个条件指令的结束。 |
| #line | 它可以让您修改编译器的行数以及（可选地）输出错误和警告的文件名。 |
| #error | 它允许从代码的指定位置生成一个错误。 |
| #warning | 它允许从代码的指定位置生成一级警告。 |
| #region | 它可以让您在使用 Visual Studio Code Editor 的大纲特性时，指定一个可展开或折叠的代码块。 |
| #endregion | 它标识着 #region 块的结束。 |

## #define 预处理器

# define 预处理器指令创建符号常量。

## define 允许您定义一个符号，这样，通过使用符号作为传递给 #if 指令的表达式，表达式将返回 true。它的语法如下：

```
#define symbol
```

下面的程序说明了这点：

```
#define PI
using System;
namespace PreprocessorDAppl
{
   class Program
   {
      static void Main(string[] args)
      {
         #if (PI)
            Console.WriteLine("PI is defined");
         #else
            Console.WriteLine("PI is not defined");
         #endif
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
PI is defined
```

# 条件指令

您可以使用 #if 指令来创建一个条件指令。条件指令用于测试符号是否为真。如果为真，编译器会执行 #if 和下一个指令之间的代码。

条件指令的语法：

```
#if symbol [operator symbol]...
```

其中，*symbol* 是要测试的符号名称。您也可以使用 true 和 false，或在符号前放置否定运算符。

运算符符号是用于评价符号的运算符。可以运算符可以是下列运算符之一：

- == (equality)
- != (inequality)
- && (and)

- || (or)

您也可以用括号把符号和运算符进行分组。条件指令用于在调试版本或编译指定配置时编译代码。一个以 **#if** 指令开始的条件指令，必须显示地以一个 **#endif** 指令终止。

下面的程序演示了条件指令的用法：

```
#define DEBUG
#define VC_V10
using System;
public class TestClass
{
   public static void Main()
   {

      #if (DEBUG && !VC_V10)
         Console.WriteLine("DEBUG is defined");
      #elif (!DEBUG && VC_V10)
         Console.WriteLine("VC_V10 is defined");
      #elif (DEBUG && VC_V10)
         Console.WriteLine("DEBUG and VC_V10 are defined");
      #else
         Console.WriteLine("DEBUG and VC_V10 are not defined");
      #endif
      Console.ReadKey();
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
DEBUG and VC_V10 are defined
```

# C# 正则表达式

正则表达式 是一种匹配输入文本的模式。.Net 框架提供了允许这种匹配的正则表达式引擎。模式由一个或多个字符、运算符和结构组成。

## 定义正则表达式

下面列出了用于定义正则表达式的各种类别的字符、运算符和结构。

- 字符转义
- 字符类
- 定位点
- 分组构造
- 限定符
- 反向引用构造
- 备用构造
- 替换
- 杂项构造

### 字符转义

正则表达式中的反斜杠字符（\）指示其后跟的字符是特殊字符，或应按原义解释该字符。

下表列出了转义字符：

| 转义字符 | 描述 | 模式 | 匹配 |
| --- | --- | --- | --- |
| **\a** | 与报警 (bell) 符 \u0007 匹配。 | \a | "Warning!" + '\u0007' 中的 "\u0007" |
| **\b** | 在字符类中，与退格键 \u0008 匹配。 | [\b]{3,} | "\b\b\b\b" 中的 "\b\b\b\b" |
| **\t** | 与制表符 \u0009 匹配。 | (\w+)\t | "Name\tAddr\t" 中的 "Name\t" 和 "Addr\t" |
| **\r** | 与回车符 \u000D 匹配。（\r 与换行符 \n 不是等效的。） | \r\n(\w+) | "\r\Hello\nWorld." 中的 "\r\nHello" |
| **\v** | 与垂直制表符 \u000B 匹配。 | [\v]{2,} | "\v\v\v" 中的 "\v\v\v" |
| **\f** | 与换页符 \u000C 匹配。 | [\f]{2,} | "\f\f\f" 中的 "\f\f\f" |
| **\n** | 与换行符 \u000A 匹配。 | \r\n(\w+) | "\r\Hello\nWorld." 中的 "\r\nHello" |
| **\e** | 与转义符 \u001B 匹配。 | \e | "\x001B" 中的 "\x001B" |
| **\ nnn** | 使用八进制表示形式指定一个字符（nnn 由二到三位数字组成）。 | \w\040\w | "a bc d" 中的 "a b" 和 "c d" |
| **\x nn** | 使用十六进制表示形式指定字符（nn 恰好由两位数字组成）。 | \w\x20\w | "a bc d" 中的 "a b" 和 "c d" |
| **\c X**<br>**\c x** | 匹配 X 或 x 指定的 ASCII 控件字符，其中 X 或 x 是控件字符的字母。 | \cC | "\x0003" 中的 "\x0003" (Ctrl-C) |
| **\u nnnn** | 使用十六进制表示形式匹配一个 Unicode 字符（由 nnnn 表示的四位数）。 | \w\u0020\w | "a bc d" 中的 "a b" 和 "c d" |
| **** | 在后面带有不识别的转义字符时，与该字符匹配。 | \d+[+-x*]\d+\d+[+-x*\d+ | "(2+2) *39" 中的 "2+2" 和 "3*9" |

## 字符类

字符类与一组字符中的任何一个字符匹配。 下表列出了字符类：

| 字符类 | 描述 | 模式 | 匹配 |
|---|---|---|---|
| **[character_group]** | 匹配 character_group 中的任何单个字符。 默认情况下，匹配区分大小写。 | [mn] | "mat" 中的 "m"，"moon" 中的 "m" 和 "n" |
| **[^character_group]** | 非：与不在 character_group 中的任何单个字符匹配。 默认情况下，character_group 中的字符区分大小写。 | [^aei] | "avail" 中的 "v" 和 "l" |
| **[ first - last ]** | 字符范围：与从 first 到 last 的范围中的任何单个字符匹配。 | (\w+)\t | "Name\tAddr\t" 中的 "Name\t" 和 "Addr\t" |
| **.** | 通配符：与除 \n 之外的任何单个字符匹配。 若要匹配原意句点字符（. 或 \u002E），您必须在该字符前面加上转义符 (\.)。 | a.e | "have" 中的 "ave"，"mate" 中的 "ate" |
| **\p{ name }** | 与 *name* 指定的 Unicode 通用类别或命名块中的任何单个字符匹配。 | \p{Lu} | "City Lights" 中的 "C" 和 "L" |
| **\P{ name }** | 与不在 *name* 指定的 Unicode 通用类别或命名块中的任何单个字符匹配。 | \P{Lu} | "City" 中的 "i"、"t" 和 "y" |
| **\w** | 与任何单词字符匹配。 | \w | "Room#1" 中的 "R"、"o"、"m" 和 "1" |
| **\W** | 与任何非单词字符匹配。 | \W | "Room#1" 中的 "#" |
| **\s** | 与任何空白字符匹配。 | \w\s | "ID A1.3" 中的 "D " |
| **\S** | 与任何非空白字符匹配。 | \s\S | "int _ctr" 中的 " " |
| **\d** | 与任何十进制数字匹配。 | \d | "4 = IV" 中的 "4" |
| **\D** | 匹配不是十进制数的任意字符。 | \D | "4 = IV" 中的 " "、"="、" "、"I" 和 "V" |

## 定位点

定位点或原子零宽度断言会使匹配成功或失败，具体取决于字符串中的当前位置，但它们不会使引擎在字符串中前进或使用字符。 下表列出了定位点：

| 断言 | 描述 | 模式 | 匹配 |
| --- | --- | --- | --- |
| **^** | 匹配必须从字符串或一行的开头开始。 | ^\d{3} | "567-777-" 中的 "567" |
| **$** | 匹配必须出现在字符串的末尾或出现在行或字符串末尾的 **\n** 之前。 | -\d{4}$ | "8-12-2012" 中的 "-2012" |
| **\A** | 匹配必须出现在字符串的开头。 | \A\w{3} | "Code-007-" 中的 "Code" |
| **\Z** | 匹配必须出现在字符串的末尾或出现在字符串末尾的 **\n** 之前。 | -\d{3}\Z | "Bond-901-007" 中的 "-007" |
| **\z** | 匹配必须出现在字符串的末尾。 | -\d{3}\z | "-901-333" 中的 "-333" |
| **\G** | 匹配必须出现在上一个匹配结束的地方。 | \G(\d) | "(1)(3)(5)7" 中的 "(1)"、 "(3)" 和 "(5)" |
| **\b** | 匹配必须出现在 **\w**（字母数字）和 **\W**（非字母数字）字符之间的边界上。 | \w | "Room#1" 中的 "R"、 "o"、 "m" 和 "1" |
| **\B** | 匹配不得出现在 **\b** 边界上。 | \Bend\w*\b | "end sends endure lender" 中的 "ends" 和 "ender" |

## 分组构造

分组构造描述了正则表达式的子表达式，通常用于捕获输入字符串的子字符串。 下表列出了分组构造：

| 分组构造 | 描述 | 模式 | 匹配 |
|---|---|---|---|
| **( subexpression )** | 捕获匹配的子表达式并将其分配到一个从零开始的序号中。 | (\w)\1 | "deep" 中的 "ee" |
| **(?< name >subexpression)** | 将匹配的子表达式捕获到一个命名组中。 | (?< double>\w)\k< double> | "deep" 中的 "ee" |
| **(?< name1 -name2 >subexpression)** | 定义平衡组定义。 | (((?'Open'()\(\))+((?'Close-Open'))\(\))+)*(?(Open)(?!))$ | "3+2^((1-3)(3-1))" 中的 "((1-3)(3-1))" |
| **(?: subexpression)** | 定义非捕获组。 | Write(?:Line)? | "Console.WriteLine()" 中的 "WriteLine" |
| **(?imnsx-imnsx:subexpression)** | 应用或禁用 *subexpression* 中指定的选项。 | A\d{2}(?i:\w+)\b | "A12xl A12XL a12xl" 中的 "A12xl" 和 "A12XL" |
| **(?= subexpression)** | 零宽度正预测先行断言。 | \w+(?=.) | "He is. The dog ran. The sun is out." 中的 "is"、 "ran" 和 "out" |
| **(?! subexpression)** | 零宽度负预测先行断言。 | \b(?!un)\w+\b | "unsure sure unity used" 中的 "sure" 和 "used" |
| **(?< =subexpression)** | 零宽度正回顾后发断言。 | (?< =19)\d{2}\b | "1851 1999 1950 1905 2003" 中的 "51" 和 "03" |
| **(?< ! subexpression)** | 零宽度负回顾后发断言。 | (?< !19)\d{2}\b | "end sends endure lender" 中的 "ends" 和 "ender" |
| **(?> subexpression)** | 非回溯（也称为"贪婪"）子表达式。 | [13579](?>A+B+) | "1ABB 3ABBC 5AB 5AC" 中的 "1ABB"、 "3ABB" 和 "5AB" |

## 限定符

限定符指定在输入字符串中必须存在上一个元素（可以是字符、组或字符类）的多少个实例才能出现匹配项。 限定符包括下表中列出的语言元素。 下表列出了限定符：

| 限定符 | 描述 | 模式 | 匹配 |
| --- | --- | --- | --- |
| ***** | 匹配上一个元素零次或多次。 | \d*.\d | ".0"、 "19.9"、 "219.9" |
| **+** | 匹配上一个元素一次或多次。 | "be+" | "been" 中的 "bee"， "bent" 中的 "be" |
| **?** | 匹配上一个元素零次或一次。 | "rai?n" | "ran"、 "rain" |
| **{ n }** | 匹配上一个元素恰好 n 次。 | ",\d{3}" | "1,043.6" 中的 ",043"， "9,876,543,210" 中的 ",876"、 ",543" 和 ",210" |
| **{ n ,}** | 匹配上一个元素至少 n 次。 | "\d{2,}" | "166"、 "29"、 "1930" |
| **{ n , m }** | 匹配上一个元素至少 n 次，但不多于 m 次。 | "\d{3,5}" | "166"， "17668"， "193024" 中的 "19302" |
| ***?** | 匹配上一个元素零次或多次，但次数尽可能少。 | \d*?.\d | ".0"、 "19.9"、 "219.9" |
| **+?** | 匹配上一个元素一次或多次，但次数尽可能少。 | "be+?" | "been" 中的 "be"， "bent" 中的 "be" |
| **??** | 匹配上一个元素零次或一次，但次数尽可能少。 | "rai??n" | "ran"、 "rain" |
| **{ n }?** | 匹配前导元素恰好 n 次。 | ",\d{3}?" | "1,043.6" 中的 ",043"， "9,876,543,210" 中的 ",876"、 ",543" 和 ",210" |
| **{ n ,}?** | 匹配上一个元素至少 n 次，但次数尽可能少。 | "\d{2,}?" | "166"、 "29" 和 "1930" |
| **{ n , m }?** | 匹配上一个元素的次数介于 n 和 m 之间，但次数尽可能少。 | "\d{3,5}?" | "166"， "17668"， "193024" 中的 "193" 和 "024" |

## 反向引用构造

反向引用允许在同一正则表达式中随后标识以前匹配的子表达式。 下表列出了反向引用构造：

| 反向引用构造 | 描述 | 模式 | 匹配 |
| --- | --- | --- | --- |
| **\ number** | 反向引用。 匹配编号子表达式的值。 | (\w)\1 | "seek" 中的 "ee" |
| **\k< name >** | 命名反向引用。 匹配命名表达式的值。 | (?< char>\w)\k< char> | "seek" 中的 "ee" |

## 备用构造

备用构造用于修改正则表达式以启用 either/or 匹配。 下表列出了备用构造：

<table>
<tr><th>备用构造</th><th>描述</th><th>模式</th><th>匹配</th></tr>
<tr><td>|</td><td>匹配以竖线 (|) 字符分隔的任何一个元素。</td><td>th(e|is|at)</td><td>"this is the day. " 中的 "the" 和 "this"</td></tr>
<tr><td>(?( expression )yes | no )</td><td>如果正则表达式模式由 expression 匹配指定，则匹配 yes；否则匹配可选的 no 部分。 expression 被解释为零宽度断言。</td><td>(?(A)A\d{2}\b|\b\d{3}\b)</td><td>"A10 C103 910" 中的 "A10" 和 "910"</td></tr>
<tr><td>(?( name )yes | no )</td><td>如果 name 或已命名或已编号的捕获组具有匹配，则匹配 yes；否则匹配可选的 no。</td><td>(?< quoted>")?(?(quoted).+?"|\S+\s)</td><td>"Dogs.jpg "Yiska playing.jpg"" 中的 Dogs.jpg 和 "Yiska playing.jpg"</td></tr>
</table>

## 替换

替换是替换模式中使用的正则表达式。 下表列出了用于替换的字符：

| 字符 | 描述 | 模式 | 替换模式 | 输入字符串 | 结果字符串 |
| --- | --- | --- | --- | --- | --- |
| **$number** | 替换按组 *number* 匹配的子字符串。 | \b(\w+)(\s)(\w+)\b | $3$2$1 | "one two" | "two one" |
| **${name}** | 替换按命名组 *name* 匹配的子字符串。 | \b(?<word1>\w+)(\s)(?<word2>\w+)\b | ${word2} ${word1} | "one two" | "two one" |
| **$** | 替换字符"$"。 | \b(\d+)\s?USD | $1 | "103 USD" | "$103" |
| **><** | 替换整个匹配项的一个副本。 | ($(\d(.+\d+)?){1}) | **>< | "$1.30" | **"$1.30"** |
| ** | 替换匹配前的输入字符串的所有文本。 | B+ | "AABBCC" | "AAAACC" | |
| **/tr>** | 替换匹配后的输入字符串的所有文本。 | B+ | /tr> | "AABBCC" | "AACCCC" |
| **$+** | 替换最后捕获的组。 | B+(C+) | $+ | "AABBCCDD" | AACCDD |
| **$_** | 替换整个输入字符串。 | B+ | $_ | "AABBCC" | "AAAABBCCCC |

杂项构造 下表列出了各种杂项构造：

| 构造 | 描述 | 实例 |
|---|---|---|
| **(?imnsx-imnsx)** | 在模式中间对诸如不区分大小写这样的选项进行设置或禁用。 | \bA(?i)b\w+\b 匹配 "ABA Able Act" 中的 "ABA" 和 "Able" |
| **(?#comment)** | 内联注释。该注释在第一个右括号处终止。 | \bA(?#Matches words starting with A)\w+\b |
| **#** [to end of line] | X 模式注释。 该注释以非转义的 # 开头，并继续到行的结尾。 | (?x)\bA\w+\b#Matches words starting with A |

# Regex 类

Regex 类用于表示一个正则表达式。 下表列出了 Regex 类中一些常用的方法：

| 方法 | 描述 |
|---|---|
| **public bool IsMatch( string input )** | 指示 Regex 构造函数中指定的正则表达式是否在指定的输入字符串中找到匹配项。 |
| **public bool IsMatch( string input, int startat )** | 指示 Regex 构造函数中指定的正则表达式是否在指定的输入字符串中找到匹配项，从字符串中指定的开始位置开始。 |
| **public static bool IsMatch( string input, string pattern )** | 指示指定的正则表达式是否在指定的输入字符串中找到匹配项。 |
| **public MatchCollection Matches( string input )** | 在指定的输入字符串中搜索正则表达式的所有匹配项。 |
| **public string Replace( string input, string replacement )** | 在指定的输入字符串中，把所有匹配正则表达式模式的所有匹配的字符串替换为指定的替换字符串。 |
| **public string[] Split( string input )** | 把输入字符串分割为子字符串数组，根据在 Regex 构造函数中指定的正则表达式模式定义的位置进行分割。 |

如需了解 Regex 类的完整的属性列表，请参阅微软的 C# 文档。

# 实例 1

下面的实例匹配了以 'S' 开头的单词：

```
using System;
using System.Text.RegularExpressions;

namespace RegExApplication
{
   class Program
   {
      private static void showMatch(string text, string expr)
      {
         Console.WriteLine("The Expression: " + expr);
         MatchCollection mc = Regex.Matches(text, expr);
         foreach (Match m in mc)
         {
            Console.WriteLine(m);
         }
      }
      static void Main(string[] args)
      {
         string str = "A Thousand Splendid Suns";

         Console.WriteLine("Matching words that start with 'S': ");
         showMatch(str, @"\bS\S*");
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Matching words that start with 'S':
The Expression: \bS\S*
Splendid
Suns
```

# 实例 2

下面的实例匹配了以 'm' 开头以 'e' 结尾的单词：

```
using System;
using System.Text.RegularExpressions;

namespace RegExApplication
{
   class Program
   {
      private static void showMatch(string text, string expr)
      {
         Console.WriteLine("The Expression: " + expr);
         MatchCollection mc = Regex.Matches(text, expr);
         foreach (Match m in mc)
         {
            Console.WriteLine(m);
         }
      }
      static void Main(string[] args)
      {
         string str = "make maze and manage to measure it";

         Console.WriteLine("Matching words start with 'm' and ends with 'e':");
         showMatch(str, @"\bm\S*e\b");
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Matching words start with 'm' and ends with 'e':
The Expression: \bm\S*e\b
make
maze
manage
measure
```

# 实例 3

下面的实例替换掉多余的空格：

```
using System;
using System.Text.RegularExpressions;

namespace RegExApplication
{
   class Program
   {
      static void Main(string[] args)
      {
         string input = "Hello   World   ";
         string pattern = "\\s+";
         string replacement = " ";
         Regex rgx = new Regex(pattern);
         string result = rgx.Replace(input, replacement);

         Console.WriteLine("Original String: {0}", input);
         Console.WriteLine("Replacement String: {0}", result);
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Original String: Hello   World
Replacement String: Hello World
```

# C# 异常处理

异常是在程序执行期间出现的问题。C# 中的异常是对程序运行时出现的特殊情况的一种响应，比如尝试除以零。

异常提供了一种把程序控制权从某个部分转移到另一个部分的方式。C# 异常处理时建立在四个关键词之上的：**try**、**catch**、**finally** 和 **throw**。

- **try**：一个 try 块标识了一个将被激活的特定的异常的代码块。后跟一个或多个 catch 块。
- **catch**：程序通过异常处理程序捕获异常。catch 关键字表示异常的捕获。
- **finally**：finally 块用于执行给定的语句，不管异常是否被抛出都会执行。例如，如果您打开一个文件，不管是否出现异常文件都要被关闭。
- **throw**：当问题出现时，程序抛出一个异常。使用 throw 关键字来完成。

## 语法

假设一个块将出现异常，一个方法使用 try 和 catch 关键字捕获异常。try/catch 块内的代码为受保护的代码，使用 try/catch 语法如下所示：

```
try
{
   // 引起异常的语句
}
catch( ExceptionName e1 )
{
   // 错误处理代码
}
catch( ExceptionName e2 )
{
   // 错误处理代码
}
catch( ExceptionName eN )
{
   // 错误处理代码
}
finally
{
   // 要执行的语句
}
```

您可以列出多个 catch 语句捕获不同类型的异常，以防 try 块在不同的情况下生成多个异常。

## C# 中的异常类

C# 异常是使用类来表示的。C# 中的异常类主要是直接或间接地派生于 **System.Exception** 类。**System.ApplicationException** 和 **System.SystemException** 类是派生于 System.Exception 类的异常类。

**System.ApplicationException** 类支持由应用程序生成的异常。所以程序员定义的异常都应派生自该类。

**System.SystemException** 类是所有预定义的系统异常的基类。

下表列出了一些派生自 Sytem.SystemException 类的预定义的异常类：

| 异常类 | 描述 |
|---|---|
| System.IO.IOException | 处理 I/O 错误。 |
| System.IndexOutOfRangeException | 处理当方法指向超出范围的数组索引时生成的错误。 |
| System.ArrayTypeMismatchException | 处理当数组类型不匹配时生成的错误。 |
| System.NullReferenceException | 处理当依从一个空对象时生成的错误。 |
| System.DivideByZeroException | 处理当除以零时生成的错误。 |
| System.InvalidCastException | 处理在类型转换期间生成的错误。 |
| System.OutOfMemoryException | 处理空闲内存不足生成的错误。 |
| System.StackOverflowException | 处理栈溢出生成的错误。 |

# 异常处理

C# 以 try 和 catch 块的形式提供了一种结构化的异常处理方案。使用这些块，把核心程序语句与错误处理语句分离开。

这些错误处理块是使用 **try**、**catch** 和 **finally** 关键字实现的。下面是一个当除以零时抛出异常的实例：

```
using System;
namespace ErrorHandlingApplication
{
    class DivNumbers
    {
        int result;
        DivNumbers()
        {
            result = 0;
        }
        public void division(int num1, int num2)
        {
            try
            {
                result = num1 / num2;
            }
            catch (DivideByZeroException e)
            {
                Console.WriteLine("Exception caught: {0}", e);
            }
            finally
            {
                Console.WriteLine("Result: {0}", result);
            }

        }
        static void Main(string[] args)
        {
            DivNumbers d = new DivNumbers();
            d.division(25, 0);
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Exception caught: System.DivideByZeroException: Attempted to divide by zero.
at ...
Result: 0
```

# 创建用户自定义异常

您也可以定义自己的异常。用户自定义的异常类是派生自 **ApplicationException** 类。下面的实例演示了这点：

```
using System;
namespace UserDefinedException
{
   class TestTemperature
   {
      static void Main(string[] args)
      {
         Temperature temp = new Temperature();
         try
         {
            temp.showTemp();
         }
         catch(TempIsZeroException e)
         {
            Console.WriteLine("TempIsZeroException: {0}", e.Message);
         }
         Console.ReadKey();
      }
   }
}
public class TempIsZeroException: ApplicationException
{
   public TempIsZeroException(string message): base(message)
   {
   }
}
public class Temperature
{
   int temperature = 0;
   public void showTemp()
   {
      if(temperature == 0)
      {
         throw (new TempIsZeroException("Zero Temperature found"));
      }
      else
      {
         Console.WriteLine("Temperature: {0}", temperature);
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
TempIsZeroException: Zero Temperature found
```

# 抛出对象

如果异常是直接或间接派生自 **System.Exception** 类，您可以抛出一个对象。您可以在 catch 块中使用 throw 语句来抛出当前的对象，如下所示：

```
Catch(Exception e)
{
   ...
   Throw e
}
```

# C# 文件的输入与输出

一个 文件 是一个存储在磁盘中带有指定名称和目录路径的数据集合。当打开文件进行读写时，它变成一个 流。

从根本上说，流是通过通信路径传递的字节序列。有两个主要的流：输入流 和 输出流。输入流用于从文件读取数据（读操作），输出流用于向文件写入数据（写操作）。

## C# I/O 类

System.IO 命名空间有各种不同的类，用于执行各种文件操作，如创建和删除文件、读取或写入文件，关闭文件等。

下表列出了一些 System.IO 命名空间中常用的非抽象类：

| I/O 类 | 描述 |
|---|---|
| BinaryReader | 从二进制流读取原始数据。 |
| BinaryWriter | 以二进制格式写入原始数据。 |
| BufferedStream | 字节流的临时存储。 |
| Directory | 有助于操作目录结构。 |
| DirectoryInfo | 用于对目录执行操作。 |
| DriveInfo | 提供驱动器的信息。 |
| File | 有助于处理文件。 |
| FileInfo | 用于对文件执行操作。 |
| FileStream | 用于文件中任何位置的读写。 |
| MemoryStream | 用于随机访问存储在内存中的数据流。 |
| Path | 对路径信息执行操作。 |
| StreamReader | 用于从字节流中读取字符。 |
| StreamWriter | 用于向一个流中写入字符。 |
| StringReader | 用于读取字符串缓冲区。 |
| StringWriter | 用于写入字符串缓冲区。 |

## FileStream 类

System.IO 命名空间中的 **FileStream** 类有助于文件的读写与关闭。该类派生自抽象类 Stream。

您需要创建一个 **FileStream** 对象来创建一个新的文件，或打开一个已有的文件。创建 **FileStream** 对象的语法如下：

```
FileStream <object_name> = new FileStream( <file_name>,
<FileMode Enumerator>, <FileAccess Enumerator>, <FileShare Enumerator>);
```

例如，创建一个 FileStream 对象 **F** 来读取名为 **sample.txt** 的文件：

```
FileStream F = new FileStream("sample.txt", FileMode.Open, FileAccess.Read, FileShare.Rea
```

| 参数 | 描述 |
|---|---|
| FileMode | **FileMode** 枚举定义了各种打开文件的方法。FileMode 枚举的成员有：**Append**：打开一个已有的文件，并将光标放置在文件的末尾。如果文件不存在，则创建文件。 ***Create***：创建一个新的文件。如果文件已存在，则删除旧文件，然后创建新文件。 ***CreateNew***：指定操作系统应创建一个新的文件。如果文件已存在，则抛出异常。 ***Open***：打开一个已有的文件。如果文件不存在，则抛出异常。 ***OpenOrCreate***：指定操作系统应打开一个已有的文件。如果文件不存在，则用指定的名称创建一个新的文件打开。 **Truncate*：打开一个已有的文件，文件一旦打开，就将被截断为零字节大小。然后我们可以向文件写入全新的数据，但是保留文件的初始创建日期。如果文件不存在，则抛出异常。 |
| FileAccess | **FileAccess** 枚举的成员有：**Read**、**ReadWrite** 和 **Write**。 |
| FileShare | **FileShare** 枚举的成员有： **Inheritable**：允许文件句柄可由子进程继承。Win32 不直接支持此功能。 **None**：谢绝共享当前文件。文件关闭前，打开该文件的任何请求（由此进程或另一进程发出的请求）都将失败。 **Read**：允许随后打开文件读取。如果未指定此标志，则文件关闭前，任何打开该文件以进行读取的请求（由此进程或另一进程发出的请求）都将失败。但是，即使指定了此标志，仍可能需要附加权限才能够访问该文件。 **ReadWrite**：允许随后打开文件读取或写入。如果未指定此标志，则文件关闭前，任何打开该文件以进行读取或写入的请求（由此进程或另一进程发出）都将失败。但是，即使指定了此标志，仍可能需要附加权限才能够访问该文件。 **Write**：允许随后打开文件写入。如果未指定此标志，则文件关闭前，任何打开该文件以进行写入的请求（由此进程或另一进过程发出的请求）都将失败。但是，即使指定了此标志，仍可能需要附加权限才能够访问该文件。 **Delete**：允许随后删除文件。 |

# 实例

下面的程序演示了 **FileStream** 类的用法：

```
using System;
using System.IO;

namespace FileIOApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            FileStream F = new FileStream("test.dat",
            FileMode.OpenOrCreate, FileAccess.ReadWrite);

            for (int i = 1; i <= 20; i++)
            {
                F.WriteByte((byte)i);
            }

            F.Position = 0;

            for (int i = 0; i <= 20; i++)
            {
                Console.Write(F.ReadByte() + " ");
            }
            F.Close();
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 -1
```

# C# 高级文件操作

上面的实例演示了 C# 中简单的文件操作。但是，要充分利用 C# System.IO 类的强大功能，您需要知道这些类常用的属性和方法。

在下面的章节中，我们将讨论这些类和它们执行的操作。请单击链接详细了解各个部分的知识：

| 主题 | 描述 |
| --- | --- |
| 文本文件的读写 | 它涉及到文本文件的读写。**StreamReader** 和 **StreamWriter** 类有助于完成文本文件的读写。 |
| 二进制文件的读写 | 它涉及到二进制文件的读写。**BinaryReader** 和 **BinaryWriter** 类有助于完成二进制文件的读写。 |
| Windows 文件系统的操作 | 它让 C# 程序员能够浏览并定位 Windows 文件和目录。 |

# C# 高级

# C# 特性（Attribute）

特性（**Attribute**）是用于在运行时传递程序中各种元素（比如类、方法、结构、枚举、组件等）的行为信息的声明性标签。您可以通过使用特性向程序添加声明性信息。一个声明性标签是通过放置在它所应用的元素前面的方括号（[ ]）来描述的。

特性（Attribute）用于添加元数据，如编译器指令和注释、描述、方法、类等其他信息。.Net 框架提供了两种类型的特性：预定义特性和自定义特性。

## 规定特性（Attribute）

规定特性（Attribute）的语法如下：

```
[attribute(positional_parameters, name_parameter = value, ...)]
element
```

特性（Attribute）的名称和值是在方括号内规定的，放置在它所应用的元素之前。positional_parameters 规定必需的信息，name_parameter 规定可选的信息。

## 预定义特性（Attribute）

.Net 框架提供了三种预定义特性：

- AttributeUsage
- Conditional
- Obsolete

### AttributeUsage

预定义特性 **AttributeUsage** 描述了如何使用一个自定义特性类。它规定了特性可应用到的项目的类型。

规定该特性的语法如下：

```
[AttributeUsage(
   validon,
   AllowMultiple=allowmultiple,
   Inherited=inherited
)]
```

其中：

- 参数 validon 规定特性可被放置的语言元素。它是枚举器 *AttributeTargets* 的值的组合。默认值是 *AttributeTargets.All*。
- 参数 *allowmultiple*（可选的）为该特性的 *AllowMultiple* 属性（property）提供一个布尔值。如果为 true，则该特性是多用的。默认值是 false（单用的）。
- 参数 *inherited*（可选的）为该特性的 *Inherited* 属性（property）提供一个布尔值。如果为 true，则该特性可被派生类继承。默认值是 false（不被继承）。

例如：

```
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Feild |
AttributeTargets.Method |
AttributeTargets.Property,
AllowMultiple = true)]
```

## Conditional

这个预定义特性标记了一个条件方法，其执行依赖于它顶的预处理标识符。

它会引起方法调用的条件编译，取决于指定的值，比如 **Debug** 或 **Trace**。例如，当调试代码时显示变量的值。

规定该特性的语法如下：

```
[Conditional(
   conditionalSymbol
)]
```

例如：

```
[Conditional("DEBUG")]
```

下面的实例演示了该特性：

```
#define DEBUG
using System;
using System.Diagnostics;
public class Myclass
{
    [Conditional("DEBUG")]
    public static void Message(string msg)
    {
        Console.WriteLine(msg);
    }
}
class Test
{
    static void function1()
    {
        Myclass.Message("In Function 1.");
        function2();
    }
    static void function2()
    {
        Myclass.Message("In Function 2.");
    }
    public static void Main()
    {
        Myclass.Message("In Main function.");
        function1();
        Console.ReadKey();
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
In Main function
In Function 1
In Function 2
```

## Obsolete

这个预定义特性标记了不应被使用的程序实体。它可以让您通知编译器丢弃某个特定的目标元素。例如，当一个新方法被用在一个类中，但是您仍然想要保持类中的旧方法，您可以通过显示一个应该使用新方法，而不是旧方法的消息，来把它标记为 obsolete（过时的）。

规定该特性的语法如下：

```
[Obsolete(
   message
)]
[Obsolete(
   message,
   iserror
)]
```

其中：

- 参数 *message*，是一个字符串，描述项目为什么过时的原因以及该替代使用什么。
- 参数 *iserror*，是一个布尔值。如果该值为 true，编译器应把该项目的使用当作一个错误。默认值是 false（编译器生成一个警告）。

下面的实例演示了该特性：

```
using System;
public class MyClass
{
   [Obsolete("Don't use OldMethod, use NewMethod instead", true)]
   static void OldMethod()
   {
      Console.WriteLine("It is the old method");
   }
   static void NewMethod()
   {
      Console.WriteLine("It is the new method");
   }
   public static void Main()
   {
      OldMethod();
   }
}
```

当您尝试编译该程序时，编译器会给出一个错误消息说明：

```
Don't use OldMethod, use NewMethod instead
```

# 创建自定义特性（Attribute）

.Net 框架允许创建自定义特性，用于存储声明性的信息，且可在运行时被检索。该信息根据设计标准和应用程序需要，可与任何目标元素相关。

创建并使用自定义特性包含四个步骤：

- 声明自定义特性
- 构建自定义特性
- 在目标程序元素上应用自定义特性
- 通过反射访问特性

最后一个步骤包含编写一个简单的程序来读取元数据以便查找各种符号。元数据是用于描述其他数据的数据和信息。该程序应使用反射来在运行时访问特性。我们将在下一章详细讨论这点。

## 声明自定义特性

一个新的自定义特性应派生自 **System.Attribute** 类。例如：

```
// 一个自定义特性 BugFix 被赋给类及其成员
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Field |
AttributeTargets.Method |
AttributeTargets.Property,
AllowMultiple = true)]

public class DeBugInfo : System.Attribute
```

在上面的代码中，我们已经声明了一个名为 *DeBugInfo* 的自定义特性。

## 构建自定义特性

让我们构建一个名为 *DeBugInfo* 的自定义特性，该特性将存储调试程序获得的信息。它存储下面的信息：

- bug 的代码编号
- 辨认该 bug 的开发人员名字
- 最后一次审查该代码的日期
- 一个存储了开发人员标记的字符串消息

我们的 *DeBugInfo* 类将带有三个用于存储前三个信息的私有属性（property）和一个用于存储消息的公有属性（property）。所以 bug 编号、开发人员名字和审查日期将是 DeBugInfo 类的必需的定位（ positional）参数，消息将是一个可选的命名（named）参数。

每个特性必须至少有一个构造函数。必需的定位（ positional）参数应通过构造函数传递。下面的代码演示了 *DeBugInfo* 类：

```
// 一个自定义特性 BugFix 被赋给类及其成员
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Field |
AttributeTargets.Method |
AttributeTargets.Property,
AllowMultiple = true)]

public class DeBugInfo : System.Attribute
{
  private int bugNo;
  private string developer;
  private string lastReview;
  public string message;

  public DeBugInfo(int bg, string dev, string d)
  {
      this.bugNo = bg;
      this.developer = dev;
      this.lastReview = d;
  }

  public int BugNo
  {
      get
      {
          return bugNo;
      }
  }
  public string Developer
  {
      get
      {
          return developer;
      }
  }
  public string LastReview
  {
      get
      {
          return lastReview;
      }
  }
  public string Message
  {
      get
      {
          return message;
      }
      set
      {
          message = value;
      }
  }
}
```

## 应用自定义特性

通过把特性放置在紧接着它的目标之前，来应用该特性：

```
[DeBugInfo(45, "Zara Ali", "12/8/2012", Message = "Return type mismatch")]
[DeBugInfo(49, "Nuha Ali", "10/10/2012", Message = "Unused variable")]
class Rectangle
{
  // 成员变量
  protected double length;
  protected double width;
  public Rectangle(double l, double w)
  {
      length = l;
      width = w;
  }
  [DeBugInfo(55, "Zara Ali", "19/10/2012",
  Message = "Return type mismatch")]
  public double GetArea()
  {
      return length * width;
  }
  [DeBugInfo(56, "Zara Ali", "19/10/2012")]
  public void Display()
  {
      Console.WriteLine("Length: {0}", length);
      Console.WriteLine("Width: {0}", width);
      Console.WriteLine("Area: {0}", GetArea());
  }
}
```

在下一章中，我们将使用 Reflection 类对象来检索这些信息。

# C# 反射（Reflection）

反射（**Reflection**） 对象用于在运行时获取类型信息。该类位于 **System.Reflection** 命名空间中，可访问一个正在运行的程序的元数据。

**System.Reflection** 命名空间包含了允许您获取有关应用程序信息及向应用程序动态添加类型、值和对象的类。

## 反射（Reflection）的用途

反射（Reflection）有下列用途：

- 它允许在运行时查看属性（attribute）信息。
- 它允许审查集合中的各种类型，以及实例化这些类型。
- 它允许延迟绑定的方法和属性（property）。
- 它允许在运行时创建新类型，然后使用这些类型执行一些任务。

## 查看元数据

我们已经在上面的章节中提到过，使用反射（Reflection）可以查看属性（attribute）信息。

**System.Reflection** 类的 **MemberInfo** 对象需要被初始化，用于发现与类相关的属性（attribute）。为了做到这点，您可以定义目标类的一个对象，如下：

```
System.Reflection.MemberInfo info = typeof(MyClass);
```

下面的程序演示了这点：

```
using System;

[AttributeUsage(AttributeTargets.All)]
public class HelpAttribute : System.Attribute
{
   public readonly string Url;

   public string Topic  // Topic 是一个命名（named）参数
   {
      get
      {
         return topic;
      }
      set
      {

         topic = value;
      }
   }

   public HelpAttribute(string url)  // url 是一个定位（positional）参数
   {
      this.Url = url;
   }

   private string topic;
}
[HelpAttribute("Information on the class MyClass")]
class MyClass
{
}

namespace AttributeAppl
{
   class Program
   {
      static void Main(string[] args)
      {
         System.Reflection.MemberInfo info = typeof(MyClass);
         object[] attributes = info.GetCustomAttributes(true);
         for (int i = 0; i < attributes.Length; i++)
         {
            System.Console.WriteLine(attributes[i]);
         }
         Console.ReadKey();

      }
   }
}
```

当上面的代码被编译和执行时，它会显示附加到类 *MyClass* 上的自定义属性：

```
HelpAttribute
```

# 实例

在本实例中，我们将使用在上一章中创建的 *DeBugInfo* 属性，并使用反射（Reflection）来读取 *Rectangle* 类中的元数据。

```
using System;
using System.Reflection;
```

```
namespace BugFixApplication
{
   // 一个自定义属性 BugFix 被赋给类及其成员
   [AttributeUsage(AttributeTargets.Class |
   AttributeTargets.Constructor |
   AttributeTargets.Field |
   AttributeTargets.Method |
   AttributeTargets.Property,
   AllowMultiple = true)]

   public class DeBugInfo : System.Attribute
   {
      private int bugNo;
      private string developer;
      private string lastReview;
      public string message;

      public DeBugInfo(int bg, string dev, string d)
      {
         this.bugNo = bg;
         this.developer = dev;
         this.lastReview = d;
      }

      public int BugNo
      {
         get
         {
            return bugNo;
         }
      }
      public string Developer
      {
         get
         {
            return developer;
         }
      }
      public string LastReview
      {
         get
         {
            return lastReview;
         }
      }
      public string Message
      {
         get
         {
            return message;
         }
         set
         {
            message = value;
         }
      }
   }
   [DeBugInfo(45, "Zara Ali", "12/8/2012",
    Message = "Return type mismatch")]
   [DeBugInfo(49, "Nuha Ali", "10/10/2012",
    Message = "Unused variable")]
   class Rectangle
   {
      // 成员变量
      protected double length;
      protected double width;
      public Rectangle(double l, double w)
      {
         length = l;
         width = w;
      }
      [DeBugInfo(55, "Zara Ali", "19/10/2012",
```

```
         Message = "Return type mismatch")]
        public double GetArea()
        {
           return length * width;
        }
        [DeBugInfo(56, "Zara Ali", "19/10/2012")]
        public void Display()
        {
           Console.WriteLine("Length: {0}", length);
           Console.WriteLine("Width: {0}", width);
           Console.WriteLine("Area: {0}", GetArea());
        }
     }//end class Rectangle

     class ExecuteRectangle
     {
        static void Main(string[] args)
        {
           Rectangle r = new Rectangle(4.5, 7.5);
           r.Display();
           Type type = typeof(Rectangle);
           // 遍历 Rectangle 类的属性
           foreach (Object attributes in type.GetCustomAttributes(false))
           {
              DeBugInfo dbi = (DeBugInfo)attributes;
              if (null != dbi)
              {
                 Console.WriteLine("Bug no: {0}", dbi.BugNo);
                 Console.WriteLine("Developer: {0}", dbi.Developer);
                 Console.WriteLine("Last Reviewed: {0}",
                      dbi.LastReview);
                 Console.WriteLine("Remarks: {0}", dbi.Message);
              }
           }

           // 遍历方法属性
           foreach (MethodInfo m in type.GetMethods())
           {
              foreach (Attribute a in m.GetCustomAttributes(true))
              {
                 DeBugInfo dbi = (DeBugInfo)a;
                 if (null != dbi)
                 {
                    Console.WriteLine("Bug no: {0}, for Method: {1}",
                          dbi.BugNo, m.Name);
                    Console.WriteLine("Developer: {0}", dbi.Developer);
                    Console.WriteLine("Last Reviewed: {0}",
                          dbi.LastReview);
                    Console.WriteLine("Remarks: {0}", dbi.Message);
                 }
              }
           }
           Console.ReadLine();
        }
     }
  }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Length: 4.5
Width: 7.5
Area: 33.75
Bug No: 49
Developer: Nuha Ali
Last Reviewed: 10/10/2012
Remarks: Unused variable
Bug No: 45
Developer: Zara Ali
Last Reviewed: 12/8/2012
Remarks: Return type mismatch
Bug No: 55, for Method: GetArea
Developer: Zara Ali
Last Reviewed: 19/10/2012
Remarks: Return type mismatch
Bug No: 56, for Method: Display
Developer: Zara Ali
Last Reviewed: 19/10/2012
Remarks:
```

# C# 属性（Property）

属性（**Property**） 是类（class）、结构（structure）和接口（interface）的命名（named）成员。类或结构中的成员变量或方法称为 域（**Field**）。属性（Property）是域（Field）的扩展，且可使用相同的语法来访问。它们使用 访问器（**accessors**） 让私有域的值可被读写或操作。

属性（Property）不会确定存储位置。相反，它们具有可读写或计算它们值的 访问器（**accessors**）。

例如，有一个名为 Student 的类，带有 age、name 和 code 的私有域。我们不能在类的范围以外直接访问这些域，但是我们可以拥有访问这些私有域的属性。

# 访问器（Accessors）

属性（Property）的访问器（**accessor**）包含有助于获取（读取或计算）或设置（写入）属性的可执行语句。访问器（accessor）声明可包含一个 get 访问器、一个 set 访问器，或者同时包含二者。例如：

```
// 声明类型为 string 的 Code 属性
public string Code
{
   get
   {
      return code;
   }
   set
   {
      code = value;
   }
}

// 声明类型为 string 的 Name 属性
public string Name
{
   get
   {
     return name;
   }
   set
   {
     name = value;
   }
}

// 声明类型为 int 的 Age 属性
public int Age
{
   get
   {
      return age;
   }
   set
   {
      age = value;
   }
}
```

# 实例

下面的实例演示了属性（Property）的用法：

```
using System;
namespace tutorialspoint
{
   class Student
   {

      private string code = "N.A";
      private string name = "not known";
      private int age = 0;

      // 声明类型为 string 的 Code 属性
      public string Code
      {
         get
         {
            return code;
         }
         set
         {
            code = value;
         }
      }
```

```
        // 声明类型为 string 的 Name 属性
        public string Name
        {
           get
           {
              return name;
           }
           set
           {
              name = value;
           }
        }

        // 声明类型为 int 的 Age 属性
        public int Age
        {
           get
           {
              return age;
           }
           set
           {
              age = value;
           }
        }
        public override string ToString()
        {
           return "Code = " + Code +", Name = " + Name + ", Age = " + Age;
        }
      }
      class ExampleDemo
      {
        public static void Main()
        {
           // 创建一个新的 Student 对象
           Student s = new Student();

           // 设置 student 的 code、name 和 age
           s.Code = "001";
           s.Name = "Zara";
           s.Age = 9;
           Console.WriteLine("Student Info: {0}", s);
           // 增加年龄
           s.Age += 1;
           Console.WriteLine("Student Info: {0}", s);
           Console.ReadKey();
         }
     }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Student Info: Code = 001, Name = Zara, Age = 9
Student Info: Code = 001, Name = Zara, Age = 10
```

# 抽象属性（Abstract Properties）

抽象类可拥有抽象属性，这些属性应在派生类中被实现。下面的程序说明了这点：

```
using System;
namespace tutorialspoint
{
```

```
    public abstract class Person
    {
       public abstract string Name
       {
          get;
          set;
       }
       public abstract int Age
       {
          get;
          set;
       }
    }
    class Student : Person
    {

       private string code = "N.A";
       private string name = "N.A";
       private int age = 0;

       // 声明类型为 string 的 Code 属性
       public string Code
       {
          get
          {
             return code;
          }
          set
          {
             code = value;
          }
       }

       // 声明类型为 string 的 Name 属性
       public override string Name
       {
          get
          {
             return name;
          }
          set
          {
             name = value;
          }
       }

       // 声明类型为 int 的 Age 属性
       public override int Age
       {
          get
          {
             return age;
          }
          set
          {
             age = value;
          }
       }
       public override string ToString()
       {
          return "Code = " + Code +", Name = " + Name + ", Age = " + Age;
       }
    }
    class ExampleDemo
    {
       public static void Main()
       {
          // 创建一个新的 Student 对象
          Student s = new Student();

          // 设置 student 的 code、name 和 age
          s.Code = "001";
```

```
            s.Name = "Zara";
            s.Age = 9;
            Console.WriteLine("Student Info:- {0}", s);
            // 增加年龄
            s.Age += 1;
            Console.WriteLine("Student Info:- {0}", s);
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Student Info: Code = 001, Name = Zara, Age = 9
Student Info: Code = 001, Name = Zara, Age = 10
```

# C# 索引器（Indexer）

索引器（**Indexer**） 允许一个对象可以像数组一样被索引。当您为类定义一个索引器时，该类的行为就会像一个 虚拟数组（**virtual array**） 一样。您可以使用数组访问运算符（[ ]）来访问该类的实例。

## 语法

一维索引器的语法如下：

```
element-type this[int index]
{
   // get 访问器
   get
   {
      // 返回 index 指定的值
   }

   // set 访问器
   set
   {
      // 设置 index 指定的值
   }
}
```

## 索引器（Indexer）的用途

索引器的行为的声明在某种程度上类似于属性（property）。就像属性（property），您可使用 **get** 和 **set** 访问器来定义索引器。但是，属性返回或设置一个特定的数据成员，而索引器返回或设置对象实例的一个特定值。换句话说，它把实例数据分为更小的部分，并索引每个部分，获取或设置每个部分。

定义一个属性（property）包括提供属性名称。索引器定义的时候不带有名称，但带有 **this** 关键字，它指向对象实例。下面的实例演示了这个概念：

```
using System;
namespace IndexerApplication
{
   class IndexedNames
   {
      private string[] namelist = new string[size];
      static public int size = 10;
      public IndexedNames()
      {
         for (int i = 0; i < size; i++)
         namelist[i] = "N. A.";
      }
      public string this[int index]
      {
         get
         {
            string tmp;

            if( index >= 0 && index <= size-1 )
            {
               tmp = namelist[index];
            }
            else
            {
               tmp = "";
            }

            return ( tmp );
         }
         set
         {
            if( index >= 0 && index <= size-1 )
            {
               namelist[index] = value;
            }
         }
      }

      static void Main(string[] args)
      {
         IndexedNames names = new IndexedNames();
         names[0] = "Zara";
         names[1] = "Riz";
         names[2] = "Nuha";
         names[3] = "Asif";
         names[4] = "Davinder";
         names[5] = "Sunil";
         names[6] = "Rubic";
         for ( int i = 0; i < IndexedNames.size; i++ )
         {
            Console.WriteLine(names[i]);
         }
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Zara
Riz
Nuha
Asif
Davinder
Sunil
Rubic
N. A.
N. A.
N. A.
```

# 重载索引器（Indexer）

索引器（Indexer）可被重载。索引器声明的时候也可带有多个参数，且每个参数可以是不同的类型。没有必要让索引器必须是整型的。C# 允许索引器可以是其他类型，例如，字符串类型。

下面的实例演示了重载索引器：

```
using System;
namespace IndexerApplication
{
   class IndexedNames
   {
      private string[] namelist = new string[size];
      static public int size = 10;
      public IndexedNames()
      {
         for (int i = 0; i < size; i++)
         {
          namelist[i] = "N. A.";
         }
      }
      public string this[int index]
      {
         get
         {
            string tmp;

            if( index >= 0 && index <= size-1 )
            {
               tmp = namelist[index];
            }
            else
            {
               tmp = "";
            }

            return ( tmp );
         }
         set
         {
            if( index >= 0 && index <= size-1 )
            {
               namelist[index] = value;
            }
         }
      }
      public int this[string name]
      {
         get
         {
            int index = 0;
```

```
            while(index < size)
            {
               if (namelist[index] == name)
               {
                return index;
               }
               index++;
            }
            return index;
         }

      }

      static void Main(string[] args)
      {
         IndexedNames names = new IndexedNames();
         names[0] = "Zara";
         names[1] = "Riz";
         names[2] = "Nuha";
         names[3] = "Asif";
         names[4] = "Davinder";
         names[5] = "Sunil";
         names[6] = "Rubic";
         // 使用带有 int 参数的第一个索引器
         for (int i = 0; i < IndexedNames.size; i++)
         {
            Console.WriteLine(names[i]);
         }
         // 使用带有 string 参数的第二个索引器
         Console.WriteLine(names["Nuha"]);
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Zara
Riz
Nuha
Asif
Davinder
Sunil
Rubic
N. A.
N. A.
N. A.
2
```

# C# 委托（Delegate）

C# 中的委托（Delegate）类似于 C 或 C++ 中函数的指针。委托（**Delegate**） 是存有对某个方法的引用的一种引用类型变量。引用可在运行时被改变。

委托（Delegate）特别用于实现事件和回调方法。所有的委托（Delegate）都派生自 **System.Delegate** 类。

## 声明委托（Delegate）

委托声明决定了可由该委托引用的方法。委托可指向一个与其具有相同标签的方法。

例如，假设有一个委托：

```
public delegate int MyDelegate (string s);
```

上面的委托可被用于引用任何一个带有一个单一的 *string* 参数的方法，并返回一个 *int* 类型变量。

声明委托的语法如下：

```
delegate <return type> <delegate-name> <parameter list>
```

## 实例化委托（Delegate）

一旦声明了委托类型，委托对象必须使用 **new** 关键字来创建，且与一个特定的方法有关。当创建委托时，传递到 **new** 语句的参数就像方法调用一样书写，但是不带有参数。例如：

```
public delegate void printString(string s);
...
printString ps1 = new printString(WriteToScreen);
printString ps2 = new printString(WriteToFile);
```

下面的实例演示了委托的声明、实例化和使用，该委托可用于引用带有一个整型参数的方法，并返回一个整型值。

```
using System;

delegate int NumberChanger(int n);
namespace DelegateAppl
{
   class TestDelegate
   {
      static int num = 10;
      public static int AddNum(int p)
      {
         num += p;
         return num;
      }

      public static int MultNum(int q)
      {
         num *= q;
         return num;
      }
      public static int getNum()
      {
         return num;
      }

      static void Main(string[] args)
      {
         // 创建委托实例
         NumberChanger nc1 = new NumberChanger(AddNum);
         NumberChanger nc2 = new NumberChanger(MultNum);
         // 使用委托对象调用方法
         nc1(25);
         Console.WriteLine("Value of Num: {0}", getNum());
         nc2(5);
         Console.WriteLine("Value of Num: {0}", getNum());
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of Num: 35
Value of Num: 175
```

# 委托的多播（**Multicasting of a Delegate**）

委托对象可使用 "+" 运算符进行合并。一个合并委托调用它所合并的两个委托。只有相同类型的委托可被合并。"-" 运算符可用于从合并的委托中移除组件委托。

使用委托的这个有用的特点，您可以创建一个委托被调用时要调用的方法的调用列表。这被称为委托的 多播（**multicasting**），也叫组播。下面的程序演示了委托的多播：

```
using System;

delegate int NumberChanger(int n);
namespace DelegateAppl
{
   class TestDelegate
   {
      static int num = 10;
      public static int AddNum(int p)
      {
         num += p;
         return num;
      }

      public static int MultNum(int q)
      {
         num *= q;
         return num;
      }
      public static int getNum()
      {
         return num;
      }

      static void Main(string[] args)
      {
         // 创建委托实例
         NumberChanger nc;
         NumberChanger nc1 = new NumberChanger(AddNum);
         NumberChanger nc2 = new NumberChanger(MultNum);
         nc = nc1;
         nc += nc2;
         // 调用多播
         nc(5);
         Console.WriteLine("Value of Num: {0}", getNum());
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of Num: 75
```

# 委托（Delegate）的用途

下面的实例演示了委托的用法。委托 *printString* 可用于引用带有一个字符串作为输入的方法，并不返回任何东西。

我们使用这个委托来调用两个方法，第一个把字符串打印到控制台，第二个把字符串打印到文件：

```
using System;
using System.IO;

namespace DelegateAppl
{
   class PrintString
   {
      static FileStream fs;
      static StreamWriter sw;
      // 委托声明
      public delegate void printString(string s);

      // 该方法打印到控制台
      public static void WriteToScreen(string str)
      {
         Console.WriteLine("The String is: {0}", str);
      }
      // 该方法打印到文件
      public static void WriteToFile(string s)
      {
         fs = new FileStream("c:\\message.txt",
         FileMode.Append, FileAccess.Write);
         sw = new StreamWriter(fs);
         sw.WriteLine(s);
         sw.Flush();
         sw.Close();
         fs.Close();
      }
      // 该方法把委托作为参数，并使用它调用方法
      public static void sendString(printString ps)
      {
         ps("Hello World");
      }
      static void Main(string[] args)
      {
         printString ps1 = new printString(WriteToScreen);
         printString ps2 = new printString(WriteToFile);
         sendString(ps1);
         sendString(ps2);
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
The String is: Hello World
```

# C# 事件（Event）

事件（**Event**） 基本上说是一个用户操作，如按键、点击、鼠标移动等等，或者是一些出现，如系统生成的通知。应用程序需要在事件发生时响应事件。例如，中断。事件是用于进程间通信。

## 通过事件使用委托

事件在类中声明且生成，且通过使用同一个类或其他类中的委托与事件处理程序关联。包含事件的类用于发布事件。这被称为 发布器（**publisher**） 类。其他接受该事件的类被称为 订阅器（**subscriber**） 类。事件使用 发布-订阅（**publisher-subscriber**） 模型。

发布器（**publisher**） 是一个包含事件和委托定义的对象。事件和委托之间的联系也定义在这个对象中。发布器（publisher）类的对象调用这个事件，并通知其他的对象。

订阅器（**subscriber**） 是一个接受事件并提供事件处理程序的对象。在发布器（publisher）类中的委托调用订阅器（subscriber）类中的方法（事件处理程序）。

## 声明事件（Event）

在类的内部声明事件，首先必须声明该事件的委托类型。例如：

```
public delegate void BoilerLogHandler(string status);
```

然后，声明事件本身，使用 **event** 关键字：

```
// 基于上面的委托定义事件
public event BoilerLogHandler BoilerEventLog;
```

上面的代码定义了一个名为 *BoilerLogHandler* 的委托和一个名为 *BoilerEventLog* 的事件，该事件在生成的时候会调用委托。

## 实例 1

```
using System;
namespace SimpleEvent
{
   using System;

   public class EventTest
   {
      private int value;

      public delegate void NumManipulationHandler();

      public event NumManipulationHandler ChangeNum;

      protected virtual void OnNumChanged()
      {
         if (ChangeNum != null)
         {
            ChangeNum();
         }
         else
         {
            Console.WriteLine("Event fired!");
         }

      }
      public EventTest(int n )
      {
         SetValue(n);
      }
      public void SetValue(int n)
      {
         if (value != n)
         {
            value = n;
            OnNumChanged();
         }
      }
   }
   public class MainClass
   {
      public static void Main()
      {
         EventTest e = new EventTest(5);
         e.SetValue(7);
         e.SetValue(11);
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Event Fired!
Event Fired!
Event Fired!
```

# 实例 2

本实例提供一个简单的用于热水锅炉系统故障排除的应用程序。当维修工程师检查锅炉时，锅炉的温度和压力会随着维修工程师的备注自动记录到日志文件中。

```
using System;
using System.IO;

namespace BoilerEventAppl
{

   // boiler 类
   class Boiler
   {
      private int temp;
      private int pressure;
      public Boiler(int t, int p)
      {
         temp = t;
         pressure = p;
      }

      public int getTemp()
      {
         return temp;
      }
      public int getPressure()
      {
         return pressure;
      }
   }
   // 事件发布器
   class DelegateBoilerEvent
   {
      public delegate void BoilerLogHandler(string status);

      // 基于上面的委托定义事件
      public event BoilerLogHandler BoilerEventLog;

      public void LogProcess()
      {
         string remarks = "O. K";
         Boiler b = new Boiler(100, 12);
         int t = b.getTemp();
         int p = b.getPressure();
         if(t > 150 || t < 80 || p < 12 || p > 15)
         {
            remarks = "Need Maintenance";
         }
         OnBoilerEventLog("Logging Info:\n");
         OnBoilerEventLog("Temparature " + t + "\nPressure: " + p);
         OnBoilerEventLog("\nMessage: " + remarks);
      }

      protected void OnBoilerEventLog(string message)
      {
         if (BoilerEventLog != null)
         {
            BoilerEventLog(message);
         }
      }
   }
   // 该类保留写入日志文件的条款
   class BoilerInfoLogger
   {
      FileStream fs;
      StreamWriter sw;
      public BoilerInfoLogger(string filename)
      {
         fs = new FileStream(filename, FileMode.Append, FileAccess.Write);
         sw = new StreamWriter(fs);
      }
      public void Logger(string info)
      {
         sw.WriteLine(info);
      }
```

```
        public void Close()
        {
            sw.Close();
            fs.Close();
        }
    }
    // 事件订阅器
    public class RecordBoilerInfo
    {
        static void Logger(string info)
        {
            Console.WriteLine(info);
        }//end of Logger

        static void Main(string[] args)
        {
            BoilerInfoLogger filelog = new BoilerInfoLogger("e:\\boiler.txt");
            DelegateBoilerEvent boilerEvent = new DelegateBoilerEvent();
            boilerEvent.BoilerEventLog += new
            DelegateBoilerEvent.BoilerLogHandler(Logger);
            boilerEvent.BoilerEventLog += new
            DelegateBoilerEvent.BoilerLogHandler(filelog.Logger);
            boilerEvent.LogProcess();
            Console.ReadLine();
            filelog.Close();
        }//end of main

    }//end of RecordBoilerInfo
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Logging info:

Temperature 100
Pressure 12

Message: O. K
```

# C# 集合（Collection）

集合（Collection）类是专门用于数据存储和检索的类。这些类提供了对栈（stack）、队列（queue）、列表（list）和哈希表（hash table）的支持。大多数集合类实现了相同的接口。

集合（Collection）类服务于不同的目的，如为元素动态分配内存，基于索引访问列表项等等。这些类创建 Object 类的对象的集合。在 C# 中，Object 类是所有数据类型的基类。

## 各种集合类和它们的用法

下面是各种常用的 **System.Collection** 命名空间的类。点击下面的链接查看细节。

| 类 | 描述和用法 |
| --- | --- |
| 动态数组（ArrayList） | 它代表了可被单独索引的对象的有序集合。 它基本上可以替代一个数组。但是，与数组不同的是，您可以使用索引在指定的位置添加和移除项目，动态数组会自动重新调整它的大小。它也允许在列表中进行动态内存分配、增加、搜索、排序各项。 |
| 哈希表（Hashtable） | 它使用键来访问集合中的元素。 当您使用键访问元素时，则使用哈希表，而且您可以识别一个有用的键值。哈希表中的每一项都有一个键/值对。键用于访问集合中的项目。 |
| 排序列表（SortedList） | 它可以使用键和索引来访问列表中的项。 排序列表是数组和哈希表的组合。它包含一个可使用键或索引访问各项的列表。如果您使用索引访问各项，则它是一个动态数组（ArrayList），如果您使用键访问各项，则它是一个哈希表（Hashtable）。集合中的各项总是按键值排序。 |
| 堆栈（Stack） | 它代表了一个后进先出的对象集合。 当您需要对各项进行后进先出的访问时，则使用堆栈。当您在列表中添加一项，称为推入元素，当您从列表中移除一项时，称为弹出元素。 |
| 队列（Queue） | 它代表了一个先进先出的对象集合。 当您需要对各项进行先进先出的访问时，则使用队列。当您在列表中添加一项，称为入队，当您从列表中移除一项时，称为出队。 |
| 点阵列（BitArray） | 它代表了一个使用值 1 和 0 来表示的二进制数组。 当您需要存储位，但是事先不知道位数时，则使用点阵列。您可以使用整型索引从点阵列集合中访问各项，索引从零开始。 |

# C# 泛型（Generic）

泛型（**Generic**） 允许您延迟编写类或方法中的编程元素的数据类型的规范，直到实际在程序中使用它的时候。换句话说，泛型允许您编写一个可以与任何数据类型一起工作的类或方法。

您可以通过数据类型的替代参数编写类或方法的规范。当编译器遇到类的构造函数或方法的函数调用时，它会生成代码来处理指定的数据类型。下面这个简单的实例将有助于您理解这个概念：

```
using System;
using System.Collections.Generic;

namespace GenericApplication
{
    public class MyGenericArray<T>
    {
        private T[] array;
        public MyGenericArray(int size)
        {
            array = new T[size + 1];
        }
        public T getItem(int index)
        {
            return array[index];
        }
        public void setItem(int index, T value)
        {
            array[index] = value;
        }
    }

    class Tester
    {
        static void Main(string[] args)
        {
            // 声明一个整型数组
            MyGenericArray<int> intArray = new MyGenericArray<int>(5);
            // 设置值
            for (int c = 0; c < 5; c++)
            {
                intArray.setItem(c, c*5);
            }
            // 获取值
            for (int c = 0; c < 5; c++)
            {
                Console.Write(intArray.getItem(c) + " ");
            }
            Console.WriteLine();
            // 声明一个字符数组
            MyGenericArray<char> charArray = new MyGenericArray<char>(5);
            // 设置值
            for (int c = 0; c < 5; c++)
            {
                charArray.setItem(c, (char)(c+97));
            }
            // 获取值
            for (int c = 0; c < 5; c++)
            {
                Console.Write(charArray.getItem(c) + " ");
            }
            Console.WriteLine();
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
0 5 10 15 20
a b c d e
```

# 泛型（Generic）的特性

使用泛型是一种增强程序功能的技术，具体表现在以下几个方面：

- 它有助于您最大限度地重用代码、保护类型的安全以及提高性能。
- 您可以创建泛型集合类。.NET 框架类库在 *System.Collections.Generic* 命名空间中包含了一些新的泛型集合类。您可以使用这些泛型集合类来替代 *System.Collections* 中的集合类。
- 您可以创建自己的泛型接口、泛型类、泛型方法、泛型事件和泛型委托。
- 您可以对泛型类进行约束以访问特定数据类型的方法。
- 关于泛型数据类型中使用的类型的信息可在运行时通过使用反射获取。

# 泛型（Generic）方法

在上面的实例中，我们已经使用了泛型类，我们可以通过类型参数声明泛型方法。下面的程序说明了这个概念：

```
using System;
using System.Collections.Generic;

namespace GenericMethodAppl
{
    class Program
    {
        static void Swap<T>(ref T lhs, ref T rhs)
        {
            T temp;
            temp = lhs;
            lhs = rhs;
            rhs = temp;
        }
        static void Main(string[] args)
        {
            int a, b;
            char c, d;
            a = 10;
            b = 20;
            c = 'I';
            d = 'V';

            // 在交换之前显示值
            Console.WriteLine("Int values before calling swap:");
            Console.WriteLine("a = {0}, b = {1}", a, b);
            Console.WriteLine("Char values before calling swap:");
            Console.WriteLine("c = {0}, d = {1}", c, d);

            // 调用 swap
            Swap<int>(ref a, ref b);
            Swap<char>(ref c, ref d);

            // 在交换之后显示值
            Console.WriteLine("Int values after calling swap:");
            Console.WriteLine("a = {0}, b = {1}", a, b);
            Console.WriteLine("Char values after calling swap:");
            Console.WriteLine("c = {0}, d = {1}", c, d);
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Int values before calling swap:
a = 10, b = 20
Char values before calling swap:
c = I, d = V
Int values after calling swap:
a = 20, b = 10
Char values after calling swap:
c = V, d = I
```

# 泛型（Generic）委托

您可以通过类型参数定义泛型委托。例如：

```
delegate T NumberChanger<T>(T n);
```

下面的实例演示了委托的使用：

```
using System;
using System.Collections.Generic;

delegate T NumberChanger<T>(T n);
namespace GenericDelegateAppl
{
    class TestDelegate
    {
        static int num = 10;
        public static int AddNum(int p)
        {
            num += p;
            return num;
        }

        public static int MultNum(int q)
        {
            num *= q;
            return num;
        }
        public static int getNum()
        {
            return num;
        }

        static void Main(string[] args)
        {
            // 创建委托实例
            NumberChanger<int> nc1 = new NumberChanger<int>(AddNum);
            NumberChanger<int> nc2 = new NumberChanger<int>(MultNum);
            // 使用委托对象调用方法
            nc1(25);
            Console.WriteLine("Value of Num: {0}", getNum());
            nc2(5);
            Console.WriteLine("Value of Num: {0}", getNum());
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of Num: 35
Value of Num: 175
```

# C# 匿名方法

我们已经提到过，委托是用于引用与其具有相同标签的方法。换句话说，您可以使用委托对象调用可由委托引用的方法。

匿名方法（**Anonymous methods**） 提供了一种传递代码块作为委托参数的技术。匿名方法是没有名称只有主体的方法。

在匿名方法中您不需要指定返回类型，它是从方法主体内的 return 语句推断的。

## 编写匿名方法的语法

匿名方法是通过使用 **delegate** 关键字创建委托实例来声明的。例如：

```
delegate void NumberChanger(int n);
...
NumberChanger nc = delegate(int x)
{
    Console.WriteLine("Anonymous Method: {0}", x);
};
```

代码块 *Console.WriteLine("Anonymous Method: {0}", x);* 是匿名方法的主体。

委托可以通过匿名方法调用，也可以通过命名方法调用，即，通过向委托对象传递方法参数。

例如：

```
nc(10);
```

## 实例

下面的实例演示了匿名方法的概念：

```
using System;

delegate void NumberChanger(int n);
namespace DelegateAppl
{
    class TestDelegate
    {
        static int num = 10;
        public static void AddNum(int p)
        {
            num += p;
            Console.WriteLine("Named Method: {0}", num);
        }

        public static void MultNum(int q)
        {
            num *= q;
            Console.WriteLine("Named Method: {0}", num);
        }
        public static int getNum()
        {
            return num;
        }

        static void Main(string[] args)
        {
            // 使用匿名方法创建委托实例
            NumberChanger nc = delegate(int x)
            {
               Console.WriteLine("Anonymous Method: {0}", x);
            };

            // 使用匿名方法调用委托
            nc(10);

            // 使用命名方法实例化委托
            nc =  new NumberChanger(AddNum);

            // 使用命名方法调用委托
            nc(5);

            // 使用另一个命名方法实例化委托
            nc =  new NumberChanger(MultNum);

            // 使用命名方法调用委托
            nc(2);
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Anonymous Method: 10
Named Method: 15
Named Method: 30
```

# C# 不安全代码

当一个代码块使用 **unsafe** 修饰符标记时，C# 允许在函数中使用指针变量。不安全代码或非托管代码是指使用了指针变量的代码块。

## 指针变量

指针 是值为另一个变量的地址的变量，即，内存位置的直接地址。就像其他变量或常量，您必须在使用指针存储其他变量地址之前声明指针。

指针变量声明的一般形式为：

```
type *var-name;
```

以下是有效的指针声明：

```
int    *ip;    /* 指向一个整数 */
double *dp;    /* 指向一个双精度数 */
float  *fp;    /* 指向一个浮点数 */
char   *ch     /* 指向一个字符 */
```

下面的实例说明了 C# 中使用了 **unsafe** 修饰符时指针的使用：

```
using System;
namespace UnsafeCodeApplication
{
    class Program
    {
        static unsafe void Main(string[] args)
        {
            int var = 20;
            int* p = &var;
            Console.WriteLine("Data is: {0} ",  var);
            Console.WriteLine("Address is: {0}",  (int)p);
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Data is: 20
Address is: 99215364
```

您也可以不用声明整个方法作为不安全代码，只需要声明方法的一部分作为不安全代码。下面的实例说明了这点。

# 使用指针检索数据值

您可以使用 **ToString()** 方法检索存储在指针变量所引用位置的数据。下面的实例演示了这点：

```
using System;
namespace UnsafeCodeApplication
{
   class Program
   {
      public static void Main()
      {
         unsafe
         {
            int var = 20;
            int* p = &var;
            Console.WriteLine("Data is: {0} " , var);
            Console.WriteLine("Data is: {0} " , p->ToString());
            Console.WriteLine("Address is: {0} " , (int)p);
         }
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Data is: 20
Data is: 20
Address is: 77128984
```

# 传递指针作为方法的参数

您可以向方法传递指针变量作为方法的参数。下面的实例说明了这点：

```
using System;
namespace UnsafeCodeApplication
{
   class TestPointer
   {
      public unsafe void swap(int* p, int *q)
      {
         int temp = *p;
         *p = *q;
         *q = temp;
      }

      public unsafe static void Main()
      {
         TestPointer p = new TestPointer();
         int var1 = 10;
         int var2 = 20;
         int* x = &var1;
         int* y = &var2;

         Console.WriteLine("Before Swap: var1:{0}, var2: {1}", var1, var2);
         p.swap(x, y);

         Console.WriteLine("After Swap: var1:{0}, var2: {1}", var1, var2);
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Before Swap: var1: 10, var2: 20
After Swap: var1: 20, var2: 10
```

# 使用指针访问数组元素

在 C# 中，数组名称和一个指向与数组数据具有相同数据类型的指针是不同的变量类型。例如，int *p 和 int[] p 是不同的类型。您可以增加指针变量 p，因为它在内存中不是固定的，但是数组地址在内存中是固定的，所以您不能增加数组 p。

因此，如果您需要使用指针变量访问数组数据，可以像我们通常在 C 或 C++ 中所做的那样，使用 **fixed** 关键字来固定指针。

下面的实例演示了这点：

```
using System;
namespace UnsafeCodeApplication
{
   class TestPointer
   {
      public unsafe static void Main()
      {
         int[]  list = {10, 100, 200};
         fixed(int *ptr = list)

         /* 显示指针中数组地址 */
         for ( int i = 0; i < 3; i++)
         {
            Console.WriteLine("Address of list[{0}]={1}",i,(int)(ptr + i));
            Console.WriteLine("Value of list[{0}]={1}", i, *(ptr + i));
         }
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Address of list[0] = 31627168
Value of list[0] = 10
Address of list[1] = 31627172
Value of list[1] = 100
Address of list[2] = 31627176
Value of list[2] = 200
```

# 编译不安全代码

为了编译不安全代码，您必须切换到命令行编译器指定 **/unsafe** 命令行。

例如，为了编译包含不安全代码的名为 prog1.cs 的程序，需在命令行中输入命令：

```
csc /unsafe prog1.cs
```

如果您使用的是 Visual Studio IDE，那么您需要在项目属性中启用不安全代码。

步骤如下：

- 通过双击资源管理器（Solution Explorer）中的属性（properties）节点，打开项目属性（**project properties**）。
- 点击 **Build** 标签页。
- 选择选项"**Allow unsafe code**"。

# C# 多线程

线程 被定义为程序的执行路径。每个线程都定义了一个独特的控制流。如果您的应用程序涉及到复杂的和耗时的操作，那么设置不同的线程执行路径往往是有益的，每个线程执行特定的工作。

线程是轻量级进程。一个使用线程的常见实例是现代操作系统中并行编程的实现。使用线程节省了 CPU 周期的浪费，同时提高了应用程序的效率。

到目前为止我们编写的程序是一个单线程作为应用程序的运行实例的单一的过程运行的。但是，这样子应用程序同时只能执行一个任务。为了同时执行多个任务，它可以被划分为更小的线程。

## 线程生命周期

线程生命周期开始于 System.Threading.Thread 类的对象被创建时，结束于线程被终止或完成执行时。

下面列出了线程生命周期中的各种状态：

- 未启动状态：当线程实例被创建但 Start 方法未被调用时的状况。
- 就绪状态：当线程准备好运行并等待 CPU 周期时的状况。
- 不可运行状态：下面的几种情况下线程是不可运行的：
    - 已经调用 Sleep 方法
    - 已经调用 Wait 方法
    - 通过 I/O 操作阻塞
- 死亡状态：当线程已完成执行或已中止时的状况。

## 主线程

在 C# 中，**System.Threading.Thread** 类用于线程的工作。它允许创建并访问多线程应用程序中的单个线程。进程中第一个被执行的线程称为主线程。

当 C# 程序开始执行时，主线程自动创建。使用 **Thread** 类创建的线程被主线程的子线程调用。您可以使用 Thread 类的 **CurrentThread** 属性访问线程。

下面的程序演示了主线程的执行：

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
    class MainThreadProgram
    {
        static void Main(string[] args)
        {
            Thread th = Thread.CurrentThread;
            th.Name = "MainThread";
            Console.WriteLine("This is {0}", th.Name);
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
This is MainThread
```

# Thread 类常用的属性和方法

下表列出了 **Thread** 类的一些常用的 属性：

| 属性 | 描述 |
| --- | --- |
| CurrentContext | 获取线程正在其中执行的当前上下文。 |
| CurrentCulture | 获取或设置当前线程的区域性。 |
| CurrentPrinciple | 获取或设置线程的当前负责人（对基于角色的安全性而言）。 |
| CurrentThread | 获取当前正在运行的线程。 |
| CurrentUICulture | 获取或设置资源管理器使用的当前区域性以便在运行时查找区域性特定的资源。 |
| ExecutionContext | 获取一个 ExecutionContext 对象，该对象包含有关当前线程的各种上下文的信息。 |
| IsAlive | 获取一个值，该值指示当前线程的执行状态。 |
| IsBackground | 获取或设置一个值，该值指示某个线程是否为后台线程。 |
| IsThreadPoolThread | 获取一个值，该值指示线程是否属于托管线程池。 |
| ManagedThreadId | 获取当前托管线程的唯一标识符。 |
| Name | 获取或设置线程的名称。 |
| Priority | 获取或设置一个值，该值指示线程的调度优先级。 |
| ThreadState | 获取一个值，该值包含当前线程的状态。 |

下表列出了 **Thread** 类的一些常用的 方法：

| 方法名 | 描述 |
| --- | --- |
| **public void Abort()** | 在调用此方法的线程上引发 ThreadAbortException，以开始终止此线程的过程。调用此方法通常会终止线程。 |
| **public static LocalDataStoreSlot AllocateDataSlot()** | 在所有的线程上分配未命名的数据槽。为了获得更好的性能，请改用以 ThreadStaticAttribute 属性标记的字段。 |
| **public static LocalDataStoreSlot AllocateNamedDataSlot( string name)** | 在所有线程上分配已命名的数据槽。为了获得更好的性能，请改用以 ThreadStaticAttribute 属性标记的字段。 |
| **public static void BeginCriticalRegion()** | 通知主机执行将要进入一个代码区域，在该代码区域内线程中止或未经处理的异常的影响可能会危害应用程序域中的其他任务。 |
| **public static void BeginThreadAffinity()** | 通知主机托管代码将要执行依赖于当前物理操作系统线程的标识的指令。 |
| **public static void EndCriticalRegion()** | 通知主机执行将要进入一个代码区域，在该代码区域内线程中止或未经处理的异常仅影响当前任务。 |
| **public static void EndThreadAffinity()** | 通知主机托管代码已执行完依赖于当前物理操作系统线程的标识的指令。 |
| **public static void FreeNamedDataSlot(string name)** | 为进程中的所有线程消除名称与槽之间的关联。为了获得更好的性能，请改用以 ThreadStaticAttribute 属性标记的字段。 |
| **public static Object GetData( LocalDataStoreSlot slot )** | 在当前线程的当前域中从当前线程上指定的槽中检索值。为了获得更好的性能，请改用以 ThreadStaticAttribute 属性标记的字段。 |
| **public static AppDomain GetDomain()** | 返回当前线程正在其中运行的当前域。 |
| **public static AppDomain GetDomainID()** | 返回唯一的应用程序域标识符。 |
| **public static LocalDataStoreSlot GetNamedDataSlot( string name )** | 查找已命名的数据槽。为了获得更好的性能，请改用以 ThreadStaticAttribute 属性标记的字段。 |

| | |
|---|---|
| **public void Interrupt()** | 中断处于 WaitSleepJoin 线程状态的线程。 |
| **public void Join()** | 在继续执行标准的 COM 和 SendMessage 消息泵处理期间，阻塞调用线程，直到某个线程终止为止。此方法有不同的重载形式。 |
| **public static void MemoryBarrier()** | 按如下方式同步内存存取：执行当前线程的处理器在对指令重新排序时，不能采用先执行 MemoryBarrier 调用之后的内存存取，再执行 MemoryBarrier 调用之前的内存存取的方式。 |
| **public static void ResetAbort()** | 取消为当前线程请求的 Abort。 |
| **public static void SetData( LocalDataStoreSlot slot, Object data )** | 在当前正在运行的线程上为此线程的当前域在指定槽中设置数据。为了获得更好的性能，请改用以 ThreadStaticAttribute 属性标记的字段。 |
| **public void Start()** | 开始一个线程。 |
| **public static void Sleep( int millisecondsTimeout )** | 让线程暂停一段时间。 |
| **public static void SpinWait( int iterations )** | 导致线程等待由 iterations 参数定义的时间量。 |
| **public static byte VolatileRead( ref byte address ) public static double VolatileRead( ref double address ) public static int VolatileRead( ref int address ) public static Object VolatileRead( ref Object address )** | 读取字段值。无论处理器的数目或处理器缓存的状态如何，该值都是由计算机的任何处理器写入的最新值。此方法有不同的重载形式。这里只给出了一些形式。 |
| **public static void VolatileWrite( ref byte address, byte value ) public static void VolatileWrite( ref double address, double value ) public static void VolatileWrite( ref int address, int value ) public static void VolatileWrite( ref Object address, Object value )** | 立即向字段写入一个值，以使该值对计算机中的所有处理器都可见。此方法有不同的重载形式。这里只给出了一些形式。 |
| **public static bool Yield()** | 导致调用线程执行准备好在当前处理器上运行的另一个线程。由操作系统选择要执行的线程。 |

# 创建线程

线程是通过扩展 Thread 类创建的。扩展的 Thread 类调用 **Start()** 方法来开始子线程的执行。

下面的程序演示了这个概念：

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
    class ThreadCreationProgram
    {
        public static void CallToChildThread()
        {
            Console.WriteLine("Child thread starts");
        }

        static void Main(string[] args)
        {
            ThreadStart childref = new ThreadStart(CallToChildThread);
            Console.WriteLine("In Main: Creating the Child thread");
            Thread childThread = new Thread(childref);
            childThread.Start();
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
In Main: Creating the Child thread
Child thread starts
```

# 管理线程

Thread 类提供了各种管理线程的方法。

下面的实例演示了 **sleep()** 方法的使用，用于在一个特定的时间暂停线程。

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
    class ThreadCreationProgram
    {
        public static void CallToChildThread()
        {
            Console.WriteLine("Child thread starts");
            // 线程暂停 5000 毫秒
            int sleepfor = 5000;
            Console.WriteLine("Child Thread Paused for {0} seconds",
                              sleepfor / 1000);
            Thread.Sleep(sleepfor);
            Console.WriteLine("Child thread resumes");
        }

        static void Main(string[] args)
        {
            ThreadStart childref = new ThreadStart(CallToChildThread);
            Console.WriteLine("In Main: Creating the Child thread");
            Thread childThread = new Thread(childref);
            childThread.Start();
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
In Main: Creating the Child thread
Child thread starts
Child Thread Paused for 5 seconds
Child thread resumes
```

# 销毁线程

**Abort()** 方法用于销毁线程。

通过抛出 **threadabortexception** 在运行时中止线程。这个异常不能被捕获，如果有 *finally* 块，控制会被送至 *finally* 块。

下面的程序说明了这点：

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
    class ThreadCreationProgram
    {
        public static void CallToChildThread()
        {
            try
            {

                Console.WriteLine("Child thread starts");
                // 计数到 10
                for (int counter = 0; counter <= 10; counter++)
                {
                    Thread.Sleep(500);
                    Console.WriteLine(counter);
                }
                Console.WriteLine("Child Thread Completed");

            }
            catch (ThreadAbortException e)
            {
                Console.WriteLine("Thread Abort Exception");
            }
            finally
            {
                Console.WriteLine("Couldn't catch the Thread Exception");
            }

        }

        static void Main(string[] args)
        {
            ThreadStart childref = new ThreadStart(CallToChildThread);
            Console.WriteLine("In Main: Creating the Child thread");
            Thread childThread = new Thread(childref);
            childThread.Start();
            // 停止主线程一段时间
            Thread.Sleep(2000);
            // 现在中止子线程
            Console.WriteLine("In Main: Aborting the Child thread");
            childThread.Abort();
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
In Main: Creating the Child thread
Child thread starts
0
1
2
In Main: Aborting the Child thread
Thread Abort Exception
Couldn't catch the Thread Exception
```

# ASP.NET

## 经典 ASP - Active Server Pages（动态服务器页面）

ASP，全称 Active Server Pages（动态服务器页面），也被称为经典 ASP，是在1998年作为微软的第一个服务器端脚本引擎推出的。

ASP 是一种使得网页中的脚本在因特网服务器上被执行的技术。

ASP 页面的文件扩展名是 .asp，通常是用 VBScript 编写的。

如果您想学习经典 ASP，请访问我们的 经典 ASP 教程。

## ASP.NET

ASP.NET 是新一代 ASP 。它与经典 ASP 是不兼容的，但 ASP.NET 可能包括经典 ASP。

ASP.NET 页面是经过编译的，这使得它们的运行速度比经典 ASP 快。

ASP.NET 具有更好的语言支持，有一大套的用户控件和基于 XML 的组件，并集成了用户身份验证。

ASP.NET 页面的扩展名是 .aspx，通常是用 VB (Visual Basic) 或者 C# (C sharp) 编写。

在 ASP.NET 中的控件可以用不同的语言（包括 C++ 和 Java）编写。

当浏览器请求 ASP.NET 文件时，ASP.NET 引擎读取文件，编译和执行脚本文件，并将结果以普通的 HTML 页面返回给浏览器。

## ASP.NET Razor

Razor 是一种将服务器代码嵌入到 ASP.NET 网页中的新的、简单的标记语法，很像经典 ASP。

Razor 具有传统的 ASP.NET 的功能，但更容易使用并且更容易学习。

## ASP.NET 编程语言

本教程介绍了以下编程语言：

- Visual Basic (VB.NET)
- C# (发音：C sharp)

# ASP.NET 服务器技术

本教程介绍了以下服务器技术

- Web Pages（Razor 语法)
- MVC（模型-视图-控制器）
- Web Forms（传统的 ASP.NET）

# ASP.NET 开发工具

ASP.NET 支持以下开发工具：

- WebMatrix
- Visual Web Developer
- Visual Studio

在本教程中，Web Pages 教程使用了 WebMatrix， MVC 教程和 Web Forms 教程使用了 Visual Web Developer。

# ASP.NET 文件扩展名

- 经典 ASP 文件的文件扩展名为 .asp
- ASP.NET 文件的文件扩展名为 .aspx
- Razor C# 语法的 ASP.NET 文件的文件扩展名为 .cshtml
- Razor VB 语法的 ASP.NET 文件的文件扩展名为 .vbhtml

# Web Pages 教程

# ASP.NET Web Pages - 教程

ASP.NET 是一个使用 HTML、CSS、JavaScript 和服务器脚本创建网页和网站的开发框架。

ASP.NET 支持三种不同的开发模式：
Web Pages（Web 页面）、MVC（Model View Controller 模型-视图-控制器）、Web Forms（Web 窗体）：

本教程介绍 **Web Pages**。

| Web Pages |
| --- |
| MVC |
| Web Forms |

## 从何入手？

多数开发人员学习一个新技术，是从查看运行实例开始的。

## 通过"运行实例"轻松学习

我们的"运行实例"工具让 Web Pages 变得更简单易学。

它在运行实例的同时显示 ASP.NET 代码和 HTML 输出。

点击"运行实例"按钮来看看它是如何工作的：

## Web Pages 实例

```
<html>
<body>
<h1>Hello Web Pages</h1>
<p>The time is @DateTime.Now</p>
</body>
</html>
```

运行实例？

## 什么是 Web Pages？

Web Pages 是三种创建 ASP.NET 网站和 Web 应用程序的编程模式中的一种。

其他两种编程模式是 Web Forms 和 MVC（Model View Controller 模型-视图-控制器）。

Web Pages 是开发 ASP.NET 网页最简单的开发模式。它提供了一种简单的方式来将 HTML、CSS、JavaScript 和服务器脚本结合起来：

- 容易学习，容易理解，容易使用
- 围绕着单一的网页创建
- 与 PHP 和经典 ASP 相似
- Visual Basic 或者 C# 的服务器脚本
- 全 HTML、CSS 和 JavaScript 控制

Web Pages 内置了数据库、视频、图形、社交媒体和其他更多的 Web Helpers，因此很容易扩展。

# Web Pages 教程

如果您刚接触 ASP.NET，建议从 Web Pages 开始学习。

在我们的 Web Pages 教程中，您将学习到如何使用 VB(Visual Basic) 或者 C#(C sharp) 最新的 Razor 服务器标记语法将 HTML、CSS、JavaScript 和服务器代码结合起来。

您也可以学习如何使用具有可编程的 Web Helpers（包括数据库、视频、图形、社交媒体等等）来扩展您的网页。

# Web Pages 实例

通过实例学习！

由于 ASP.NET 代码是在服务器上执行的，您不能在您的浏览器中查看代码。您只能看到普通的 HTML 页面输出。

在 w3cschool.cc 中，每个实例都会把隐藏的 ASP.NET 代码显示出来，这将让您更容易地理解它是如何工作的。

Web Pages 实例

# Web Pages 参考手册

在本教程的最后，您将看到一套完整的 ASP.NET 参考手册，介绍了对象、组件、属性和方法。

Web Pages 参考手册

# 使用 WebMatrix

在本教程中，我们使用了 WebMatrix 。

WebMatrix 是一个简单但功能强大的，由微软专门为 Web Pages 量身定做的，免费的 ASP.NET 开发工具。

WebMatrix 包含：

- Web Pages 实例和模板
- 一种 Web 服务器语言（VB 或者 C# 的 Razor 服务器标记语法）
- 一种 Web 服务器（IIS Express）
- 一种数据库服务器（SQL Server Compact）
- 一个完整的 Web 开发框架（ASP.NET）

通过使用 WebMatrix， 您可以从一个空的网站和一个空白页面开始开发，或者您也可以使用"Web 应用程序库"中的开源应用程序进行二次开发。PHP 和 ASP.NET 应用程序很多都是开源的，比如 Umbraco、DotNetNuke、Drupal、Joomla、WordPress 等等。WebMatrix 也有内置安全性、搜索引擎优化和网络出版工具。

使用 WebMatrix 开发的技术和代码可以无缝地转化为完全专业化的 ASP.NET 应用程序。

如果您想尝试使用 WebMatrix， 请点击下面的链接进行安装：

http://www.microsoft.com/web/gallery/install.aspx?appid=WebMatrix

# ASP.NET Web Pages - 添加 Razor 代码

在本教程中，我们将使用 C# 和 Visual Basic 代码的 Razor 标记。

## 什么是 Razor ？

- Razor 是一种将基于服务器的代码添加到网页中的标记语法
- Razor 具有传统 ASP.NET 标记的功能，但更容易使用并且更容易学习
- Razor 是一种服务器端标记语法，与 ASP 和 PHP 很像
- Razor 支持 C# 和 Visual Basic 编程语言

## 添加 Razor 代码

请记住上一章实例中的网页：

```
<!DOCTYPE html>

<html lang="en">
<head>
<meta charset="utf-8" />
<title>Web Pages Demo</title>
</head>
<body>
<h1>Hello Web Pages</h1>
</body>
</html>
```

现在向实例中添加一些 Razor 代码：

## 实例

```
<!DOCTYPE html>

<html lang="en">
<head>
<meta charset="utf-8" />
<title>Web Pages Demo</title>
</head>
<body>
<h1>Hello Web Pages</h1>
<p>The time is @DateTime.Now</p>
</body>
</html>
```

运行实例？

该页面中包含普通的 HTML 标记，除此之外，还添加了一个 @ 标识的 Razor 代码。

Razor 代码能够在服务器上实时地完成多有的动作，并将结果显示出来。（您可以指定格式化选项，否则只会显示默认项。）

# 主要的 Razor C# 语法规则

- Razor 代码块包含在 @{ ... } 中
- 内联表达式（变量和函数）以 @ 开头
- 代码语句用分号结束
- 变量使用 var 关键字声明
- 字符串用引号括起来
- C# 代码区分大小写
- C# 文件的扩展名是 .cshtml

# C# 实例

```
<!-- Single statement block -->
@{ var myMessage = "Hello World"; }

<!-- Inline expression or variable -->
<p>The value of myMessage is: @myMessage</p>

<!-- Multi-statement block -->
@{
var greeting = "Welcome to our site!";
var weekDay = DateTime.Now.DayOfWeek;
var greetingMessage = greeting + " Today is: " + weekDay;
}
<p>The greeting is: @greetingMessage</p>
```

运行实例？

# 主要的 Razor VB 语法规则

- Razor 代码块包含在 @Code ... End Code 中
- 内联表达式（变量和函数）以 @ 开头
- 变量使用 Dim 关键字声明
- 字符串用引号括起来
- VB 代码不区分大小写
- VB 文件的扩展名是 .vbhtml

# 实例

```
<!-- Single statement block -->
@Code dim myMessage = "Hello World" End Code

<!-- Inline expression or variable -->
<p>The value of myMessage is: @myMessage</p>

<!-- Multi-statement block -->
@Code
dim greeting = "Welcome to our site!"
dim weekDay = DateTime.Now.DayOfWeek
dim greetingMessage = greeting & " Today is: " & weekDay
End Code

<p>The greeting is: @greetingMessage</p>
```

运行实例？

# 更多关于 C# 和 Visual Basic

如果您想学习更多关于 Razor、C#、Visual Basic 编程语言，请查看本教程的 Razor 部分。

# ASP.NET Web Pages - 页面布局

通过 Web Pages，创建一个布局一致的网站是很容易的事。

## 一致的外观

在因特网上，您会发现很过网站都具有一致的外观和风格：

- 每个页面有相同的头部
- 每个页面有相同的底部
- 每个页面有相同的样式和布局

通过 Web Pages，您能非常高效地做到这点。您可以把重复使用的内容块（比如页面头部和底部）写在一个单独的文件中。

您还可以使用布局模板（布局文件）为站点的所有网页定义一致的布局。

## Content Blocks（内容块）

许多网站都有一些内容是被显示在站点的每个页面中（比如页面头部和底部）。

通过 Web Pages，您可以使用 **@RenderPage()** 方法从不同的文件导入内容。

内容块（来自另一个文件）能被导入网页中的任何地方。内容块可以包含文本，标记和代码，就像任何普通的网页一样。

将共同的头部和底部写成单独的文件，这样会帮您节省大量的工作。您不必在每个页面中书写相同的内容，当内容有变动时，您只要修改头部或者底部文件，就可以看到站点中的每个页面的相应内容都已更新。

以下显示了它在代码中是如何呈现的：

## 实例

```
<html>
<body>
@RenderPage("header.cshtml")
<h1>Hello Web Pages</h1>
<p>This is a paragraph</p>
@RenderPage("footer.cshtml")
</body>
</html>
```

运行实例？

# Layout Page（布局页）

在上一部分，您看到了，想在多个网页中显示相同内容是非常容易的。

另一种创建一致外观的方法是使用布局页。一个布局页包含了网页的结构，而不是内容。当一个网页（内容页）链接到布局页，它会根据布局页（模板）的结构进行显示。

布局页中使用 **@RenderBody()** 方法嵌入内容页，除此之外，它与一个正常的网页没有什么差别。

每个内容页都必须以布局指令开始。

以下显示了它在代码中是如何呈现的：

## 布局页：

```
<html>
<body>
<p>This is header text</p>
@RenderBody()
<p>&copy; 2012 W3CSchool. All rights reserved.</p>
</body>
</html>
```

## 任何网页：

```
@{Layout="Layout.cshtml";}

<h1>Welcome to W3CSchool.cc</h1>

<p>
Lorem ipsum dolor sit amet, consectetur adipisicing elit,sed do eiusmod tempor incididunt
</p>
```

运行实例？

# D.R.Y. - Don't Repeat Yourself（不要自我重复）

通过 Content Blocks（内容块）和 Layout Pages（布局页）这两个 ASP.NET 工具，您可以让您的 Web 应用程序显示一致的外观。

这两个工具能帮您节省大量的工作，您不必再每个页面上重复相同的信息。集中的标记、样式和代码让您的 Web 应用程序更易于管理，更易于维护。

## 防止文件被浏览

在 ASP.NET 中，文件的名称以下划线开头，可以防止这些文件在网上被浏览。

如果您不想让您的内容块或者布局页被您的用户看到，可以重命名这些文件：

_header.cshtm

_footer.cshtml

_Layout.cshtml

## 隐藏敏感信息

在 ASP.NET 中，隐藏敏感信息（数据库密码、电子邮件密码等等）最通用的方法是将这些信息保存在一个名为"_AppStart"的单独的文件中。

# _AppStart.cshtml

```
@{
WebMail.SmtpServer = "mailserver.example.com";
WebMail.EnableSsl = true;
WebMail.UserName = "username@example.com";
WebMail.Password = "your-password";
WebMail.From = "your-name-here@example.com";
}
```

# ASP.NET Web Pages - 文件夹

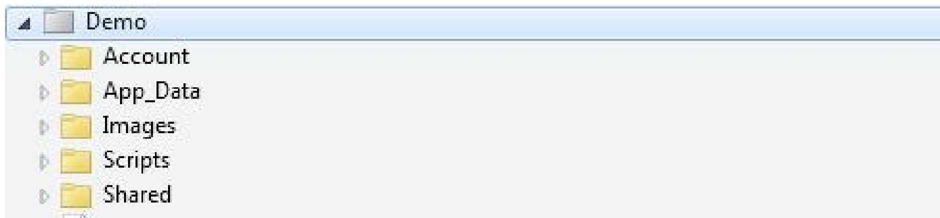
本章介绍有关文件夹和文件夹路径的知识。

在本章中，您将学到：

- 逻辑文件夹结构和物理文件夹结构
- 虚拟名称和物理名称
- Web URL 和 Web 路径

## 逻辑文件夹结构

下面是典型的 ASP.NET 网站文件夹结构：




- "Account" 文件夹包含登录和安全文件
- "App_Data" 文件夹包含数据库和数据文件
- "Images" 文件夹包含图片
- "Scripts" 文件夹包含浏览器脚本
- "Shared" 文件夹包含公共的文件（比如布局和样式文件）

## 物理文件夹结构

在上述网站中的"Images"文件夹在计算机上的物理文件夹结构可能如下：

C:\Documents\MyWebSites\Demo\Images

## 虚拟名称和物理名称

以上面的例子为例：

网站图片的虚拟名称可能是"Images/pic31.jpg"。

对应的物理名称是"C:\Documents\MyWebSites\Demo\Images\pic31.jpg"。

# URL 和路径

URL 是用来访问网站中的文件： http://www.w3cschool.cc/html/html-tutorial.html

URL 对应于服务器上的物理文件：C:\MyWebSites\w3cschool\html\html-tutorial.html

虚拟路径是物理路径的一种简写表示。如果您使用虚拟路径，当您更改域名或者将您的网页移到其他服务器上时，您可以不用更新路径。

| URL | http://www.w3cschool.cc/html/html-tutorial.html |
| --- | --- |
| 服务器名称 | w3cschool |
| 虚拟路径 | /html/html-tutorial.html |
| 物理路径 | C:\MyWebSites\w3cschool\html\html-tutorial.html |

磁盘驱动器的根目录如下书写 C:，但是网站的根目录是 / （斜线）。

Web 文件夹的虚拟路径通常是与物理文件夹不相同。

在您的代码中，根据您的编码需要决定使用物理路径和和虚拟路径。

ASP.NET 文件夹路径有 3 种工具：~ 运算符、Server.MapPath 方法 和 Href 方法。

## ~ 运算符

使用 ~ 运算符，在编程代码中规定虚拟路径。

如果您使用 ~ 运算符，在您的站点迁移到其他不同的文件夹或者位置时，您可以不用更改您的任何代码：

```
var myImagesFolder = "~/images";
var myStyleSheet = "~/styles/StyleSheet.css";
```

# Server.MapPath 方法

Server.MapPath 方法将虚拟路径（/index.html）转换成服务器能理解的物理路径（C:\Documents\MyWebSites\Demo\default.html）。

当您需要打开服务器上的数据文件时，您可以使用这个方法（只有提供完整的物理路径才能访问数据文件）：

```
var pathName = "~/dataFile.txt";
var fileName = Server.MapPath(pathName);
```

在本教程的下一章中，您会学到更多关于读取（和写入）服务器上的数据文件的知识。

# Href 方法

Href 方法将代码中的使用的路径转换成浏览器可以理解的路径（浏览器无法理解 ~ 运算符）。

您可以使用 Href 方法创建资源（比如图像文件 和 CSS 文件）的路径。

一般会在 HTML 中的 <a>、<img> 和 <link> 元素中使用此方法:

```
@{var myStyleSheet = "~/Shared/Site.css";}

<!-- This creates a link to the CSS file. -->
<link rel="stylesheet" type="text/css" href="@Href(myStyleSheet)" />

<!-- Same as : -->
<link rel="stylesheet" type="text/css" href="/Shared/Site.css" />
```

Href 方法是 WebPage 对象的一种方法。

# ASP.NET Web Pages - 全局页面

本章介绍全局页面 AppStart 和 PageStart。

## 在 Web 启动之前：_AppStart

大多数的服务器端代码是写在个人网页里边。例如，如果网页中包含输入表单，那么这个网页通常包含用来读取表单数据的服务器端代码。

然而，您可以通过在您的站点根目录下创建一个名为 _AppStart 的页面，这样在站点启动之前可以先启动代码执行。如果存在此页面，ASP.NET 会在站点中其它页面被请求时，优先运行这个页面。

_AppStart 的典型用途是启动代码和初始化全局数值（比如计数器和全局名称）。

注释 **1**：_AppStart 的文件扩展名与您的网页一致，比如：_AppStart.cshtml。

注释 **2**：_AppStart 有下划线前缀。因此，这些文件不可以直接浏览。

## 在每一个页面之前：_PageStart

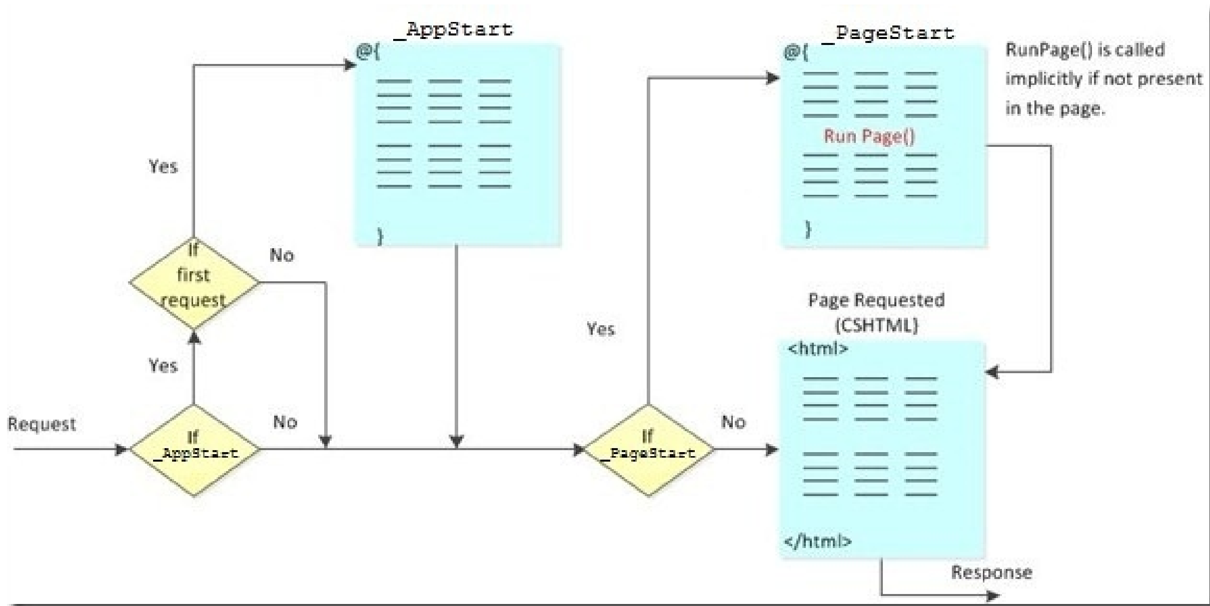
就像 _AppStart 在您的站点启动之前就运行一样，您可以编写在每个文件夹中的任何页面之前运行的代码。

对于您网站中的每个文件夹，您可以添加一个名为 _PageStart 的文件。

_PageStart 的典型用途是为一个文件夹中的所有页面设置布局页面，或者在运行某个页面之前检查用户是否已经登录。

## 它是如何工作的？

下图显示了它是如何工作的：




当接收到一个请求时，ASP.NET 会首先检查 _AppStart 是否存在。 如果 _AppStart 存在且这是站点接收到的第一个请求，则运行 _AppStart。

然后 ASP.NET 检查 _PageStart 是否存在。如果 _PageStart 存在，则在其它被请求的页面运行之前先运行 _PageStart。

您可以在 _PageStart 中调用 RunPage() 来指定被请求页面的运行位置。否则，默认情况下，被请求页面是在 _PageStart 运行之后才被运行。

# ASP.NET Web Pages - HTML 表单

表单是 HTML 文档中放置输入控件（文本框、复选框、单选按钮、下拉列表）的部分。

## 创建一个 HTML 输入页面

## Razor 实例

```
<html>
<body>
@{
if (IsPost) {
string companyname = Request["companyname"];
string contactname = Request["contactname"];
<p>You entered: <br />
Company Name: @companyname <br />
Contact Name: @contactname </p>
}
else
{
<form method="post" action="">
Company Name:<br />
<input type="text" name="CompanyName" value="" /><br />
Contact Name:<br />
<input type="text" name="ContactName" value="" /><br /><br />
<input type="submit" value="Submit" class="submit" />
</form>
}
}
</body>
</html>
```

运行实例？

## Razor 实例 - 显示图像

假设在您的图像文件夹中有 3 张图像，您想根据用户的选择动态地显示图像。

这可以通过一段简单的 Razor 代码来实现。

如果在您的网站的图像文件夹中有一个名为 "Photo1.jpg" 的图像，您可以使用 HTML 的 <img> 元素来显示图像，如下所示：



下面的例子演示了如何显示用户从下列列表中选择的图像：

## Razor 实例

```
@{
var imagePath="";
if (Request["Choice"] != null)
{imagePath="images/" + Request["Choice"];}
}
<!DOCTYPE html>
<html>
<body>
<h1>Display Images</h1>
<form method="post" action="">
I want to see:
<select name="Choice">
<option value="Photo1.jpg">Photo 1</option>
<option value="Photo2.jpg">Photo 2</option>
<option value="Photo3.jpg">Photo 3</option>
</select>
<input type="submit" value="Submit" />
@if (imagePath != "")
{
<p>
<img src="@imagePath" alt="Sample" />
</p>
}
</form>
</body>
</html>
```

运行实例？

# 实例解释

服务器创建了一个叫 **imagePath** 的变量。

HTML 页面有一个名为 **Choice** 的下拉列表（<select> 元素）。它允许用户根据自己的意愿选择一个名称（如 **Photo 1**），当页面被提交到 Web 服务器时，则传递了一个文件名（如 **Photo1.jpg**）。

Razor 代码通过 **Request["Choice"]** 读取 Choice 的值。如果通过代码构建的图像路径（images/Photo1.jpg）有效，就把图像路径赋值给变量 **imagePath**。

在 HTML 页面中，<img> 元素用来显示图像。当页面显示时，src 属性用来设置 imagePath 变量的值。

<img> 元素是在一个 if 块中，这是为了防止显示没有名称的图像，比如页面第一次被加载显示的时候。

# ASP.NET Web Pages - 对象

Web Pages 经常是跟对象有关的。

## Page 对象

您已经看到了一些在使用的 Page 对象方法：

```
@RenderPage("header.cshtml")

@RenderBody()
```

在前面的章节中，您已经看到了两个 Page 对象属性（isPost 和 Request）：

```
If (isPost) {

if (Request["Choice"] != null {
```

## 某些 Page 对象方法

| 方法 | 描述 |
| --- | --- |
| href | 使用指定的值创建 URL。 |
| RenderBody() | 呈现不在布局页命名区域的内容页的一部分。 |
| RenderPage(*page*) | 在另一个页面中呈现某一个页面的内容。 |
| RenderSection(*section*) | 呈现布局页命名区域的内容。 |
| Write(*object*) | 将对象作为 HTML 编码字符串写入。 |
| WriteLiteral | 写入对象时优先不使用 HTML 编码。 |

## 某些 Page 对象属性

| 属性 | 描述 |
| --- | --- |
| isPost | 如果客户端使用的 HTTP 数据传输方法是 POST 请求，则返回 true。 |
| Layout | 获取或者设置布局页面的路径。 |
| Page | 提供了对页面和布局页之间共享的数据的类似属性访问。 |
| Request | 为当前的 HTTP 请求获取 HttpRequest 对象。 |
| Server | 获取 HttpServerUtility 对象，该对象提供了网页处理方法。 |

# Page 对象的 Page 属性

Page 对象的 Page 属性，提供了对页面和布局页之间共享的数据的类似属性访问。

您可以对 Page 属性使用（添加）您自己的属性：

- Page.Title
- Page.Version
- Page.anythingyoulike

页面属性是非常有用的。例如，在内容文件中设置页面标题，并在布局文件中使用：

# Home.cshtml

```
@{
Layout="~/Shared/Layout.cshtml";
Page.Title="Home Page"
}

<h1>Welcome to W3CSchool.cc</h1>

<h2>Web Site Main Ingredients</h2>

<p>A Home Page (Default.cshtml)</p>
<p>A Layout File (Layout.cshtml)</p>
<p>A Style Sheet (Site.css)</p>
```

# Layout.cshtml

```
<!DOCTYPE html>
<html>
<head>
<title>@Page.Title</title>
</head>
<body>
@RenderBody()
</body>
</html
```

# ASP.NET Web Pages - 文件

本章介绍有关使用文本文件的知识。

## 使用文本文件

在前面的章节中，我们已经了解到网页数据是存储在数据库中的。

您也可以把站点数据存储在文本文件中。

用来存储数据的文本文件通常被称为平面文件。常见的文本文件格式是 .txt、.xml 和 .csv（逗号分隔值）。

在本章中，您将学习到：

- 如何从文本文件中读取并显示数据

## 手动添加一个文本文件

在下面的例子中，您将需要一个文本文件。

在您的网站上，如果没有 App_Data 文件夹，请创建一个。在 App_Data 文件夹中，创建一个名为 Persons.txt 的文件。

添加以下内容到文件中：

# Persons.txt

```
George,Lucas
Steven,Spielberg
Alfred,Hitchcock
```

## 显示文本文件中的数据

下面的实例演示了如何显示一个文本文件中的数据：

## 实例

```
@{
var dataFile = Server.MapPath("~/App_Data/Persons.txt");
Array userData = File.ReadAllLines(dataFile);
}

<!DOCTYPE html>
<html>
<body>

<h1>Reading Data from a File</h1>
@foreach (string dataLine in userData)
{
foreach (string dataItem in dataLine.Split(','))
{@dataItem <text> </text>}
<br />
}
</body>
</html>
```

运行实例？

# 实例解释

使用 **Server.MapPath** 找到确切的文本文件的路径。

使用 **File.ReadAllLines** 打开文本文件，并读取文件中的所有行到一个数组中。

数组中的每个数据行中的数据项的数据被显示。

# 显示 Excel 文件中的数据

使用 Microsoft Excel，您可以将一个电子表格保存为一个逗号分隔的文本文件（.csv 文件）。此时，电子表格中的每一行保存为一个文本行，每个数据列由逗号分隔。

in可以使用上面的实例读取一个 Excel .csv 文件（只需将文件名改成相应的 Excel 文件的名称）。

# ASP.NET Web Pages - 帮助器

Web 帮助器大大简化了 Web 开发和常见的编程任务。

## ASP.NET 帮助器

ASP.NET 帮助器是通过几行简单的 Razor 代码即可访问的组件。

您可以使用存放在 .cshtml 文件中的 Razor 语法构建自己的帮助器，或者使用内建的 ASP.NET 帮助器。

在本教程接下来的章节中，您将学到如何使用 Razor 帮助器。

下面是一些有用的 Razor 帮助器的简短说明：

## WebGrid 帮助器

WebGrid 帮助器简化了显示数据的方式：

- 自动创建一个 HTML 表格来显示数据
- 支持不同的格式化选项
- 支持数据分页显示（第一页、下一页、上一页、最后一页）
- 支持通过点击列表标题进行排序

## Chart 帮助器

"Chart 帮助器" 能显示不同类型的带有多种格式化选项和标签的图表图像。

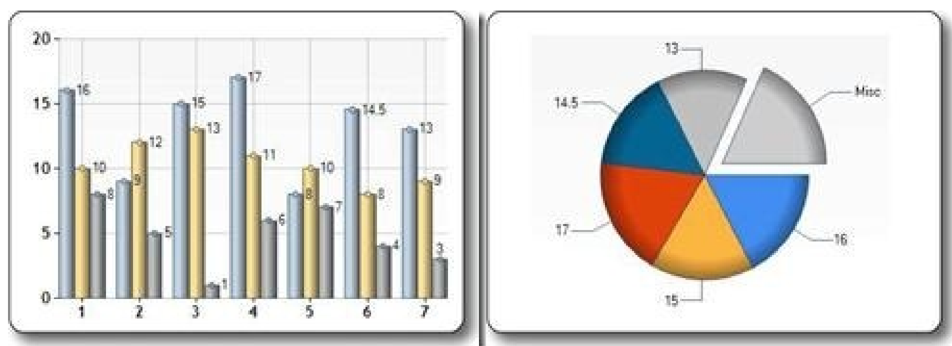



Chart 帮助器显示的数据来源可以是数组、数据库或者文件。

## WebMail 帮助器

WebMail 帮助器提供了使用SMTP（Simple Mail Transfer Protocol 简单邮件传输协议）发送电子邮件的功能。

# WebImage 帮助器

WebImage 帮助器提供了管理网页中图像的功能。

关键词：翻转、旋转、缩放、水印。

# 第三方帮助器

通过 Razor，您可以利用内建的或者第三方的帮助器来简化电子邮件、数据库、多媒体、社交网络以及很多其他问题（如导航和的网络安全）的使用。

# 安装帮助器

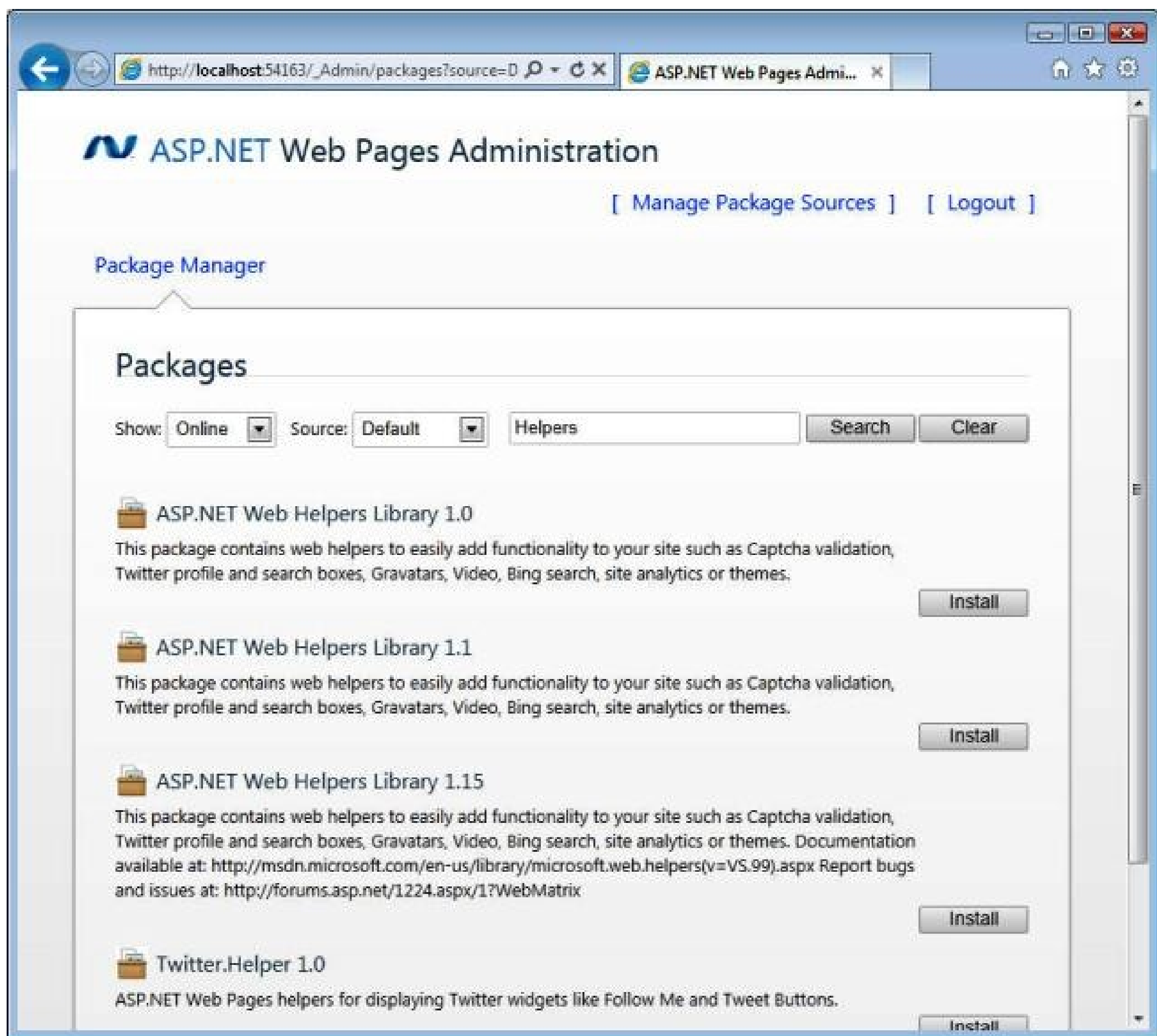
WebMatrix 已经包含了一些帮助器，您还可以手动安装其他的帮助器。

在 w3cschool.cc 的 WebPages 帮助器参考手册中，您可以看到一个便捷的参考手册，包含了内建帮助器和其他可以通过手动安装附加到 ASP.NET Web Helpers Library 工具包中的帮助器。

如果您在 WebMatrix 中创建了一个网站，请按照下面的步骤安装帮助器：

1. 在 WebMatrix 中，打开 **Site** 工作区。
2. 点击 **Web Pages Administration**。
3. 使用密码 * 登录到 Web Pages Administration。
4. 使用 搜索区 搜索帮助器。
5. 点击 **Install** 安装您所需的帮助器。

（* 如果您是第一次使用 Web Pages Administration，会提示您创建一个密码。）


http://localhost:54163/_Admin/packages?source=D
ASP.NET Web Pages Admi...
ASP.NET Web Pages Administration
[ Manage Package Sources ]
[ Logout ]
Package Manager
Packages
Show: Online
Source: Default
Helpers
Search
Clear
ASP.NET Web Helpers Library 1.0
This package contains web helpers to easily add functionality to your site such as Captcha validation, Twitter profile and search boxes, Gravatars, Video, Bing search, site analytics or themes.
Install
ASP.NET Web Helpers Library 1.1
This package contains web helpers to easily add functionality to your site such as Captcha validation, Twitter profile and search boxes, Gravatars, Video, Bing search, site analytics or themes.
Install
ASP.NET Web Helpers Library 1.15
This package contains web helpers to easily add functionality to your site such as Captcha validation, Twitter profile and search boxes, Gravatars, Video, Bing search, site analytics or themes. Documentation available at: http://msdn.microsoft.com/en-us/library/microsoft.web.helpers(v=VS.99).aspx Report bugs and issues at: http://forums.asp.net/1224.aspx/1?WebMatrix
Install
Twitter.Helper 1.0
ASP.NET Web Pages helpers for displaying Twitter widgets like Follow Me and Tweet Buttons.

# ASP.NET Web Pages - WebGrid 帮助器

WebGrid - 众多有用的 ASP.NET Web 帮助器之一。

## 自己写的 HTML

在前面的章节中，您使用 Razor 代码显示数据库数据，所有的 HTML 标记都是手写的：

## 数据库实例

```
@{
var db = Database.Open("SmallBakery");
var selectQueryString = "SELECT * FROM Product ORDER BY Name";
}
<html>
<body>
<h1>Small Bakery Products</h1>
<table>
<tr>
<th>Id</th>
<th>Product</th>
<th>Description</th>
<th>Price</th>
</tr>
@foreach(var row in db.Query(selectQueryString))
{
<tr>
<td>@row.Id</td>
<td>@row.Name</td>
<td>@row.Description</td>
<td align="right">@row.Price</td>
</tr>
}
</table>
</body>
</html>
```

运行实例？

## 使用 WebGrid 帮助器

WebGrid 帮助器提供了一种更简单的显示数据的方法。

WebGrid 帮助器：

- 自动创建一个 HTML 表格来显示数据
- 支持不同的格式化选项
- 支持数据分页显示
- 支持通过点击列表标题进行排序

# WebGrid 实例

```
@{
var db = Database.Open("SmallBakery") ;
var selectQueryString = "SELECT * FROM Product ORDER BY Id";
var data = db.Query(selectQueryString);
var grid = new WebGrid(data);
}
<html>
<head>
<title>Displaying Data Using the WebGrid Helper</title>
</head>
<body>
<h1>Small Bakery Products</h1>
<div id="grid">
@grid.GetHtml()
</div>
</body>
</html>
```

运行实例？

# ASP.NET Web Pages - Chart 帮助器

Chart 帮助器 - 众多有用的 ASP.NET Web 帮助器之一。

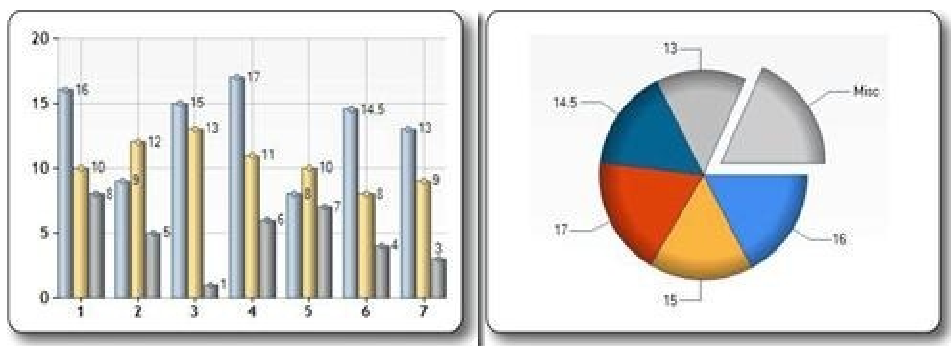
## Chart 帮助器

在前面的章节中，您已经学习了如何使用 ASP.NET 的 "帮助器"。

前面已经介绍了如何使用 "WebGrid 帮助器" 在网格中显示数据。

本章介绍如何使用 "Chart 帮助器" 以图形化的形式显示数据。

"Chart 帮助器" 可以创建不同类型的带有多种格式化选项和标签的图表图像。它可以创建面积图、条形图、柱形图、折线图、饼图等标准图表，也可以创建像股票图表这样的更专业的图表。




在图表中显示的数据可以是来自一个数组，一个数据库，或者一个文件中的数据。

## 根据数组创建图表

下面的实例显示了根据数组数据显示图表所需的代码：

## 实例

```
@{
var myChart = new Chart(width: 600, height: 400)
.AddTitle("Employees")
.AddSeries(chartType: "column",
xValue: new[] { "Peter", "Andrew", "Julie", "Mary", "Dave" },
yValues: new[] { "2", "6", "4", "5", "3" })
.Write();
}
```

运行实例？

- **new Chart** 创建一个新的图表对象并且设置它的宽度和高度
- **AddTitle** 方法指定了图表的标题
- **AddSeries** 方法向图表中增加数据
- **chartType** 参数定义图表的类型
- **xValue** 参数定义 x 轴的名称
- **yValues** 参数定义 y 轴的名称
- **Write()** 方法显示图表

# 根据数据库创建图表

您可以执行一个数据库查询，然后使用查询结果中的数据来创建一个图表：

# 实例

```
@{
var db = Database.Open("SmallBakery");
var dbdata = db.Query("SELECT Name, Price FROM Product");
var myChart = new Chart(width: 600, height: 400)
.AddTitle("Product Sales")
.DataBindTable(dataSource: dbdata, xField: "Name")
.Write();
}
```

运行实例？

- **var db = Database.Open** 打开数据库（将数据库对象赋值给变量 db）
- **var dbdata = db.Query** 执行数据库查询并保存结果在 dbdata 中
- **new Chart** 创建一个新的图表对象并且设置它的宽度和高度
- **AddTitle** 方法指定了图表的标题
- **DataBindTable** 方法将数据源绑定到图表
- **Write()** 方法显示图表

除了使用 DataBindTable 方法之外，另一种方法是使用 AddSeries（见前面的实例）。DataBindTable 更容易使用，但是 AddSeries 更加灵活，因为您可以更明确地指定图表和数据：

# 实例

```
@{
var db = Database.Open("SmallBakery");
var dbdata = db.Query("SELECT Name, Price FROM Product");
var myChart = new Chart(width: 600, height: 400)
.AddTitle("Product Sales")
.AddSeries(chartType:"Pie",
xValue: dbdata, xField: "Name",
yValues: dbdata, yFields: "Price")
.Write();
}
```

运行实例？

# 根据 XML 数据创建图表

第三种创建图表的方法是使用 XML 文件作为图表的数据：

## 实例

```
@using System.Data;

@{
var dataSet = new DataSet();
dataSet.ReadXmlSchema(Server.MapPath("data.xsd"));
dataSet.ReadXml(Server.MapPath("data.xml"));
var dataView = new DataView(dataSet.Tables[0]);
var myChart = new Chart(width: 600, height: 400)
.AddTitle("Sales Per Employee")
.AddSeries("Default", chartType: "Pie",
xValue: dataView, xField: "Name",
yValues: dataView, yFields: "Sales")
.Write();}
}
```

运行实例？

# ASP.NET Web Pages - WebMail 帮助器

WebMail 帮助器 - 众多有用的 ASP.NET Web 帮助器之一。

## WebMail 帮助器

WebMail 帮助器让发送邮件变得更简单，它按照 SMTP（Simple Mail Transfer Protocol 简单邮件传输协议）从 Web 应用程序发送邮件。

## 前提：电子邮件支持

为了演示如何使用电子邮件，我们将创建一个输入页面，让用户提交一个页面到另一个页面，并发送一封关于支持问题的邮件。

## 第一：编辑您的 AppStart 页面

如果在本教程中您已经创建了 Demo 应用程序，那么您已经有一个名为 _AppStart.cshtml 的页面，内容如下：

## _AppStart.cshtml

```
@{
WebSecurity.InitializeDatabaseConnection("Users", "UserProfile", "UserId", "Email", true)
}
```

要启动 WebMail 帮助器，向您的 AppStart 页面中增加如下所示的 WebMail 属性：

## _AppStart.cshtml

```
@{
WebSecurity.InitializeDatabaseConnection("Users", "UserProfile", "UserId", "Email", true)
WebMail.SmtpServer = "smtp.example.com";
WebMail.SmtpPort = 25;
WebMail.EnableSsl = false;
WebMail.UserName = "support@example.com";
WebMail.Password = "password-goes-here";
WebMail.From = "john@example.com";
}
```

属性解释：

**SmtpServer:** 用于发送电子邮件的 SMTP 服务器的名称。

**SmtpPort:** 服务器用来发送 SMTP 事务（电子邮件）的端口。

**EnableSsl:** 如果服务器使用 SSL（Secure Socket Layer 安全套接层）加密，则值为 true。

**UserName:** 用于发送电子邮件的 SMTP 电子邮件账户的名称。

**Password:** SMTP 电子邮件账户的密码。

**From:** 在发件地址栏显示的电子邮件（通常与 UserName 相同）。

## 第二：创建一个电子邮件输入页面

接着创建一个输入页面，并将它命名为 Email_Input：

# Email_Input.cshtml

```
<!DOCTYPE html>
<html>
<body>
<h1>Request for Assistance</h1>

<form method="post" action="EmailSend.cshtml">
<label>Username:</label>
<input type="text name="customerEmail" />
<label>Details about the problem:</label>
<textarea name="customerRequest" cols="45" rows="4"></textarea>
<p><input type="submit" value="Submit" /></p>
</form>

</body>
</html>
```

输入页面的目的是手机信息，然后提交数据到可以将信息作为电子邮件发送的一个新的页面。

## 第三：创建一个电子邮件发送页面

接着创建一个用来发送电子邮件的页面，并将它命名为 Email_Send：

# Email_Send.cshtml

```
@{ // Read input
var customerEmail = Request["customerEmail"];
var customerRequest = Request["customerRequest"];
try
{
// Send email
WebMail.Send(to:"someone@example.com", subject: "Help request from - " + customerEmail, b
}
catch (Exception ex )
{
<text>@ex</text>
}
}
```

想了解更多关于 ASP.NET Web Pages 应用程序发送电子邮件的信息，请查阅：WebMail 对象参考手册。

# ASP.NET Web Pages - PHP

PHP 开发人员请注意，Web Pages 可以用 PHP 编写。

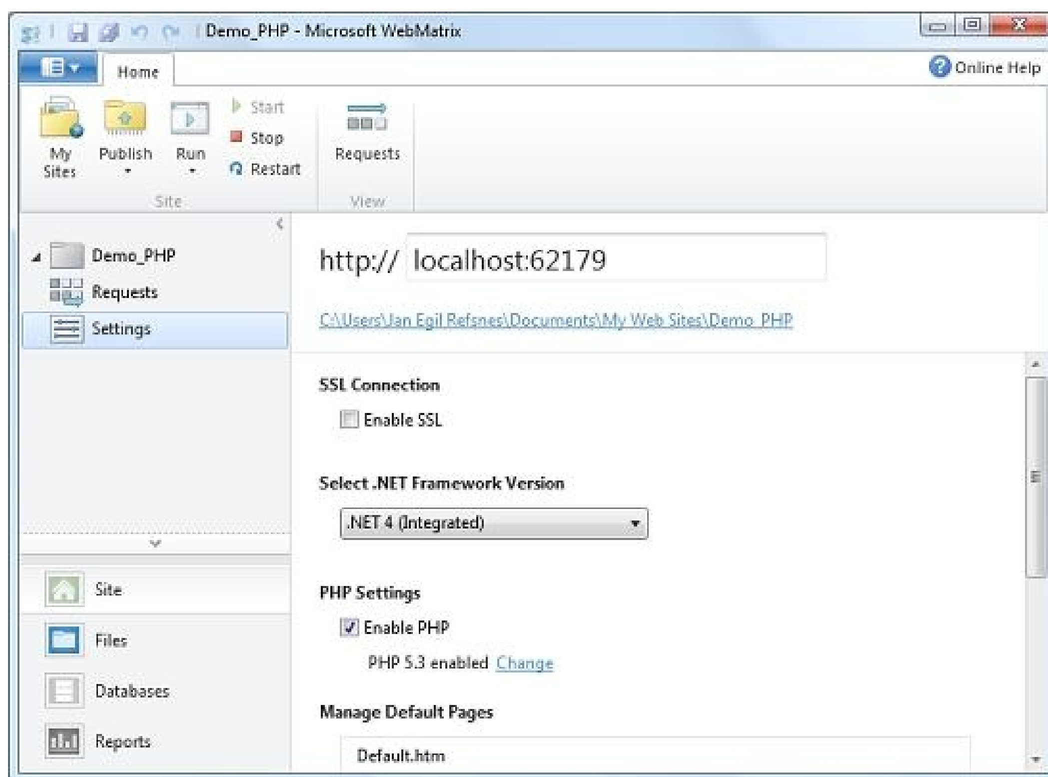
## WebMatrix 支持 PHP

乍一看，认为 WebMatrix 只支持微软的技术。这是不正确的。在 WebMatrix 中，您能编写完整的 PHP 应用程序。

## 创建一个 PHP 站点

在ASP.NET Web Pages - 创建一个网站章节中，您已经创建了一个名为 "Demo" 的空网站，带有一个类型为 "CSHTML" 的空页面。

重复一遍创建的过程，创建一个名为 "Demo_PHP" 的空站点，勾选 "Enable PHP"（如下图所示），创建一个 PHP 类型的空白页，并将它命名 "index.php"，这样您就创建好了您的第一个 PHP 站点。




## 创建一个 PHP 页面

将下面的代码复制到 "index.php" 文件中：

# index.php

```
<!DOCTYPE html>
<html>
<body>

<?php
phpinfo();
?>

</body>
</html>
```

运行文件，看看 PHP 页面的演示。

# ASP.NET Web Pages - 发布网站

学习如何在不使用 WebMatrix 的情况下发布 Web Pages 应用程序。

## 在不使用 WebMatrix 的情况下发布您的应用程序

通过在 WebMatrix（或者 Visual Studio）中使用发布命令，可以发布一个 ASP.NET Web Pages 应用程序到远程服务器上。

此功能会复制所有您的应用程序文件、cshtml页面、图像以及用于 Web Pages、Razor、Helpers、SQL Server Compact（如果使用数据库）所有必需的 DLL 文件。

有时您不想使用 WebMatrix 发布您的应用程序。也许是因为您的托管服务提供商只支持 FTP，也许您已经有一个基于经典 ASP 的网站，也许您想自己复制所有的文件，也许您想使用 Front Page、Expression Web 等其他一些发布软件。

您会遇到问题吗？是的，会的。但是您有办法解决它。

要执行网站复制，您必须知道如何引用正确的文件，哪些 DLL 文件需要复制，并在何处存储它们。

请按照下列步骤操作：

## 1. 使用最新版本的 ASP.NET

在您继续操作之前，请确保您的主机运行的是最新版的 ASP.NET（4.0 或者 4.5）。

## 2. 复制 Web 文件夹

从您的开发计算机上复制您的网站（所有文件夹和内容）到远程主机（服务器）上的应用程序文件夹中。

>  如果您的应用程序中包含数据，不要复制数据（详见下面的第 4 点）。

## 3. 复制 DLL 文件

确保您的远程主机上的 bin 文件夹中包含了和您开发计算机上相同的 dll 文件。

复制 bin 文件夹之后，它应该包含以下文件：

Microsoft.Web.Infrastructure.dll
NuGet.Core.dll
System.Web.Helpers.dll
System.Web.Razor.dll
System.Web.WebPages.Administration.dll
System.Web.WebPages.Deployment.dll
System.Web.WebPages.dll
System.Web.WebPages.Razor.dll
WebMatrix.Data.dll
WebMatrix.WebData

# 4. 复制您的数据

如果您的应用程序包含数据或者数据库。例如 SQL Server Compact 数据库（在 App_Data 文件夹中的一个 .sdf 文件），请考虑以下几点：

您是否希望发布您的测试数据到远程服务器上？

大多数时候一般是不希望。

如果在您的开发计算机上有测试数据，它将覆盖您的远程主机上的生产数据。

如果您一定要复制 SQL 数据库（.sdf 文件），那么您应该删除数据库中的所有数据，然后从您的开发计算机上复制一个空的 .sdf 文件到服务器上。

就是这样。**GOOD LUCK**！

# Razor 教程

# ASP.NET Razor - 标记

Razor 不是一种编程语言。它是服务器端的标记语言。

## 什么是 Razor？

Razor 是一种标记语法，可以让您将基于服务器的代码（Visual Basic 和 C#）嵌入到网页中。

基于服务器的代码可以在网页传送给浏览器时，创建动态 Web 内容。当一个网页被请求时，服务器在返回页面给浏览器之前先执行页面中的基于服务器的代码。通过服务器的运行，代码能执行复杂的任务，比如进入数据库。

Razor 是基于 ASP.NET 的，是为创建 Web 应用程序而设计的。它具有传统 ASP.NET 的功能，但更容易使用并且更容易学习。

## Razor 语法

Razor 使用了与 PHP 和经典 ASP 相似的语法。

Razor：

```
<ul>
@for (int i = 0; i < 10; i++) {
<li>@i</li>
}
</ul>
```

PHP：

```
<ul>
<?php
for ($i = 0; $i < 10; $i++) {
echo("<li>$i</li>");
}
?>
</ul>
```

Web Forms（经典 ASP）：

```
<ul>
<% for (int i = 0; i < 10; i++) { %>
<li><% =i %></li>
<% } %>
</ul>
```

# Razor 帮助器

ASP.NET 帮助器是通过几行简单的 Razor 代码即可访问的组件。

您可以使用 Razor 语法构建自己的帮助器，或者使用内建的 ASP.NET 帮助器。

下面是一些有用的 Razor 帮助器的简短说明：

- Web Grid（Web 网格）
- Web Graphics（Web 图形）
- Google Analytics（Google 分析）
- Facebook Integration（Facebook 集成）
- Twitter Integration（Twitter 集成）
- Sending Email（发送电子邮件）
- Validation（验证）

# Razor 编程语言

Razor 支持 C# (C sharp) 和 VB (Visual Basic)。

# ASP.NET Razor - C# 和 VB 代码语法

Razor 同时支持 C# (C sharp) 和 VB (Visual Basic)。

## 主要的 Razor C# 语法规则

- Razor 代码块包含在 @{ ... } 中
- 内联表达式（变量和函数）以 @ 开头
- 代码语句用分号结束
- 变量使用 var 关键字声明
- 字符串用引号括起来
- C# 代码区分大小写
- C# 文件的扩展名是 .cshtml

## C# 实例

```
<!-- Single statement block -->
@{ var myMessage = "Hello World"; }

<!-- Inline expression or variable -->
<p>The value of myMessage is: @myMessage</p>

<!-- Multi-statement block -->
@{
var greeting = "Welcome to our site!";
var weekDay = DateTime.Now.DayOfWeek;
var greetingMessage = greeting + " Here in Huston it is: " + weekDay;
}
<p>The greeting is: @greetingMessage</p>
```

运行实例？

## 主要的 Razor VB 语法规则

- Razor 代码块包含在 @Code ... End Code 中
- 内联表达式（变量和函数）以 @ 开头
- 变量使用 Dim 关键字声明
- 字符串用引号括起来
- VB 代码不区分大小写
- VB 文件的扩展名是 .vbhtml

# 实例

```
<!-- Single statement block -->
@Code dim myMessage = "Hello World" End Code

<!-- Inline expression or variable -->
<p>The value of myMessage is: @myMessage</p>

<!-- Multi-statement block -->
@Code
dim greeting = "Welcome to our site!"
dim weekDay = DateTime.Now.DayOfWeek
dim greetingMessage = greeting & " Here in Huston it is: " & weekDay
End Code

<p>The greeting is: @greetingMessage</p>
```

运行实例？

# 它是如何工作的？

Razor 是一种将服务器代码嵌入在网页中的简单的编程语法。

Razor 语法是基于 ASP.NET 框架，专门用于创建 Web 应用程序的部分 Microsoft.NET 框架。

Razor 语法支持所有 ASP.NET 的功能，但是使用的是一种简化语法，对初学者而言更容易学习，对专家而言更有效率的。

Razor 网页可以被描述成带一下两种类型内容的 HTML 网页：HTML 内容和 Razor 代码。

当服务器读取页面时，它首先运行 Razor 代码，然后再发送 HTML 页面到浏览器。在服务器上执行的代码能够执行一些在浏览器上不能完成的任务，比如，访问服务器数据库。服务器代码能创建动态的 HTML 内容，然后发送到浏览器。从浏览器上看，服务器代码生成的 HTML 与静态的 HTML 内容没有什么不同。

带 Razor 语法的 ASP.NET 网页有特殊的文件扩展名 cshtml（Razor C#）或者 vbhtml（Razor VB）。

# 使用对象

服务器编码往往涉及到对象。

"Date" 对象是一个典型的内置的 ASP.NET 对象，但对象也可以是自定义的，一个网页，一个文本框，一个文件，一个数据库记录，等等。

对象有用于执行的方法。一个数据库记录可能有一个 "Save" 方法，一个图像对象可能有一个 "Rotate" 方法，一个电子邮件对象可能有一个 "Send" 方法，等等。

对象也有用于描述各自特点的属性。一个数据库记录可能有 FirstName 和 LastName 属性。

ASP.NET Date 对象有一个 Now 属性（写成 Date.Now），Now 属性有一个 Day 属性（写成 Date.Now.Day）。下面实例演示了如何访问 Data 对象的一些属性：

# 实例

```
<table border="1">
<tr>
<th width="100px">Name</th>
<td width="100px">Value</td>
</tr>
<tr>
<td>Day</td><td>@DateTime.Now.Day</td>
</tr>
<tr>
<td>Hour</td><td>@DateTime.Now.Hour</td>
</tr>
<tr>
<td>Minute</td><td>@DateTime.Now.Minute</td>
</tr>
<tr>
<td>Second</td><td>@DateTime.Now.Second</td>
</tr>
</td>
</table>
```

运行实例？

# If 和 Else条件

动态网页的一个重要特点是，您可以根据条件决定做什么。

做到这一点的常用方法是使用 if ... else 语句：

# 实例

```
@{
var txt = "";
if(DateTime.Now.Hour > 12)
{txt = "Good Evening";}
else
{txt = "Good Morning";}
}
<html>
<body>
<p>The message is @txt</p>
</body>
</html>
```

运行实例？

# 读取用户输入

动态网页的另一个重要特点是，您可以读取用户输入。

输入是通过 Request[] 功能读取的，并且传送输入数据是经过 IsPost 条件判断的：

# 实例

```
@{
var totalMessage = "";
if(IsPost)
{
var num1 = Request["text1"];
var num2 = Request["text2"];
var total = num1.AsInt() + num2.AsInt();
totalMessage = "Total = " + total;
}
}
<html>
<body style="background-color: beige; font-family: Verdana, Arial;">
<form action="" method="post">
<p><label for="text1">First Number:</label><br>
<input type="text" name="text1" /></p>
<p><label for="text2">Second Number:</label><br>
<input type="text" name="text2" /></p>
<p><input type="submit" value=" Add " /></p>
</form>
<p>@totalMessage</p>
</body>
</html>
```

运行实例？

# ASP.NET Razor - C# 变量

变量是用来存储数据的命名实体。

## 变量

变量是用来存储数据的。

一个变量的名称必须以字母字符开头，并且不能包含空格或者保留字符。

一个变量可以是一个指定的类型，表示它所存储的数据类型。string 变量存储字符串值（"Welcome to W3CSchool.cc"），integer 变量存储数字值（103），date 变量存储日期值，等等。

变量使用 var 关键字声明，或通过使用类型（如果您想声明类型）声明，但是 ASP.NET 通常能自动确定数据类型。

## 实例

```
// Using the var keyword:
var greeting = "Welcome to W3CSchool.cc";
var counter = 103;
var today = DateTime.Today;

// Using data types:
string greeting = "Welcome to W3CSchool.cc";
int counter = 103;
DateTime today = DateTime.Today;
```

## 数据类型

下面列出了常用的数据类型：

| 类型 | 描述 | 实例 |
| --- | --- | --- |
| int | 整数（全数字） | 103, 12, 5168 |
| float | 浮点数 | 3.14, 3.4e38 |
| decimal | 十进制数字（高精度） | 1037.196543 |
| bool | 布尔值 | true, false |
| string | 字符串 | "Hello W3CSchool.cc", "John" |

# 运算符

运算符告诉 ASP.NET 在表达式中执行什么样的命令。

C# 语言支持多种运算符。下面列出了常用的运算符：

<table>
<tr><th>运算符</th><th>描述</th><th>实例</th></tr>
<tr><td>=</td><td>给一个变量赋值。</td><td>i=6</td></tr>
<tr><td>+<br>-<br>*<br>/</td><td>加上一个值或者一个变量。 减去一个值或者一个变量。 乘以一个值或者一个变量。 除以一个值或者一个变量。</td><td>i=5+5 i=5-5 i=5*5 i=5/5</td></tr>
<tr><td>+=<br>-=</td><td>变量递增。 变量递减。</td><td>i += 1 i -= 1</td></tr>
<tr><td>==</td><td>相等。如果值相等则返回 true。</td><td>if (i==10)</td></tr>
<tr><td>!=</td><td>不等。如果值不等则返回 true。</td><td>if (i!=10)</td></tr>
<tr><td><<br>><br><=<br>>=</td><td>小于。 大于。 小于等于。 大于等于。</td><td>if (i<10) if (i>10)<br>if (i<=10) if (i>=10)</td></tr>
<tr><td>+</td><td>连接字符串（一系列互相关联的事物）。</td><td>"w3" + "schools"</td></tr>
<tr><td>.</td><td>点号。分隔对象和方法。</td><td>DateTime.Hour</td></tr>
<tr><td>()</td><td>圆括号。将值进行分组。</td><td>(i+5)</td></tr>
<tr><td>()</td><td>圆括号。传递参数。</td><td>x=Add(i,5)</td></tr>
<tr><td>[]</td><td>中括号。访问数组或者集合的值。</td><td>name[3]</td></tr>
<tr><td>!</td><td>非。真/假取反。</td><td>if (!ready)</td></tr>
<tr><td>&&<br>||</td><td>逻辑与。 逻辑或。</td><td>if (ready && clear)<br>if (ready || clear)</td></tr>
</table>

# 转换数据类型

从一种数据类型转换到另一种数据类型，有时候是很有用的。

最常见的例子是将字符串输入转换为另一种类型，如整数或者日期。

一般规则下，都是将用户输入看做字符串处理，即使用户输入了数字。因此数值输入必须被转换成数字，然后才能将其用于计算。

下面列出了常用的转换方法：

| 方法 | 描述 | 实例 |
| --- | --- | --- |
| AsInt() IsInt() | 转换字符串为整数。 | if (myString.IsInt()) {myInt=myString.AsInt();} |
| AsFloat() IsFloat() | 转换字符串为浮点数。 | if (myString.IsFloat()) {myFloat=myString.AsFloat();} |
| AsDecimal() IsDecimal() | 转换字符串为十进制数。 | if (myString.IsDecimal()) {myDec=myString.AsDecimal();} |
| AsDateTime() IsDateTime() | 转换字符串为 ASP.NET DateTime 类型。 | myString="10/10/2012"; myDate=myString.AsDateTime(); |
| AsBool() IsBool() | 转换字符串为布尔值。 | myString="True"; myBool=myString.AsBool(); |
| ToString() | 转换任何数据类型为字符串。 | myInt=1234; myString=myInt.ToString(); |

# ASP.NET Razor - C# 循环和数组

语句在循环中会被重复执行。

## For 循环

如果您需要重复执行相同的语句，您可以设定一个循环。

如果您知道要循环的次数，您可以使用 **for** 循环。这种类型的循环在向上计数或向下计数时特别有用：

## 实例

```
<html>
<body>
@for(var i = 10; i < 21; i++)
{<p>Line @i</p>}
</body>
</html>
```

运行实例？

## For Each 循环

如果您使用的是集合或者数组，您会经常用到 **for each** 循环。

集合是一组相似的对象，for each 循环可以遍历集合直到完成。

下面的实例中，遍历 ASP.NET Request.ServerVariables 集合。

## 实例

```
<html>
<body>
<ul>
@foreach (var x in Request.ServerVariables)
{<li>@x</li>}
</ul>
</body>
</html>
```

运行实例？

# While 循环

**while** 循环是一个通用的循环。

while 循环以 while 关键字开始，后面紧跟着括号，您可以在括号里规定循环将持续多久，然后是重复执行的代码块。

while 循环通常会设定一个递增或者递减的变量用来计数。

下面的实例中，+= 运算符在每执行一次循环时给变量 i 的值加 1。

## 实例

```
<html>
<body>
@{
var i = 0;
while (i < 5)
{
i += 1;
<p>Line #@i</p>
}
}
</body>
</html>
```

运行实例？

# 数组

当您要存储多个相似变量但又不想为每个变量都创建一个独立的变量时，可以使用数组来存储：

## 实例

```
@{
string[] members = {"Jani", "Hege", "Kai", "Jim"};
int i = Array.IndexOf(members, "Kai")+1;
int len = members.Length;
string x = members[2-1];
}
<html>
<body>
<h3>Members</h3>
@foreach (var person in members)
{
<p>@person</p>
}
<p>The number of names in Members are @len</p>
<p>The person at position 2 is @x</p>
<p>Kai is now in position @i</p>
</body>
</html>
```

运行实例？

# ASP.NET Razor - C# 逻辑条件

编程逻辑：根据条件执行代码。

## If 条件

C# 允许根据条件执行代码。

使用 **if** 语句来判断条件。根据判断结果，if 语句返回 true 或者 false：

- if 语句开始一个代码块
- 条件写在括号里
- 如果条件为真，大括号内的代码被执行

## 实例

```
@{var price=50;}
<html>
<body>
@if (price>30)
{
<p>The price is too high.</p>
}
</body>
</html>
```

运行实例？

## Else 条件

if 语句可以包含 **else** 条件。

else 条件定义了当条件为假时被执行的代码。

## 实例

```
@{var price=20;}
<html>
<body>
@if (price>30)
{
<p>The price is too high.</p>
}
else
{
<p>The price is OK.</p>
}
</body>
</html>
```

运行实例？

注释：在上面的实例中，如果第一个条件为真，if 块的代码将会被执行。else 条件覆盖了除 if 条件之外的"其他所有情况"。

# Else If 条件

多个条件判断可以使用 **else if** 条件：

# 实例

```
@{var price=25;}
<html>
<body>
@if (price>=30)
{
<p>The price is high.</p>
}
else if (price>20 && price<30)
{
<p>The price is OK.</p>
}
else
{
<p>The price is low.</p>
}
</body>
</html>
```

运行实例？

在上面的实例中，如果第一个条件为真，if 块的代码将会被执行。

如果第一个条件不为真且第二个条件为真，else if 块的代码将会被执行。

else if 条件的数量不受限制。

如果 if 和 else if 条件都不为真，最后的 else 块（不带条件）覆盖了"其他所有情况"。

# Switch 条件

**switch** 块可以用来测试一些单独的条件：

## 实例

```
@{
var weekday=DateTime.Now.DayOfWeek;
var day=weekday.ToString();
var message="";
}
<html>
<body>
@switch(day)
{
case "Monday":
message="This is the first weekday.";
break;
case "Thursday":
message="Only one day before weekend.";
break;
case "Friday":
message="Tomorrow is weekend!";
break;
default:
message="Today is " + day;
break;
}
<p>@message</p>
</body>
</html>
```

运行实例？

测试值（day）是写在括号中。每个单独的测试条件都有一个以分号结束的 case 值和以 break 语句结束的任意数量的代码行。如果测试值与 case 值相匹配，相应的代码行被执行。

switch 块有一个默认的情况（default:），当所有的指定的情况都不匹配时，它覆盖了"其他所有情况"。

# ASP.NET Razor - VB 变量

变量是用来存储数据的命名实体。

## 变量

变量是用来存储数据的。

一个变量的名称必须以字母字符开头，并且不能包含空格或者保留字符。

一个变量可以是一个指定的类型，表示它所存储的数据类型。string 变量存储字符串值（"Welcome to W3CSchool.cc"），integer 变量存储数字值（103），date 变量存储日期值，等等。

变量使用 Dim 关键字声明，或通过使用类型（如果您想声明类型）声明，但是 ASP.NET 通常能自动确定数据类型。

## 实例

```
// Using the Dim keyword:
Dim greeting = "Welcome to W3CSchool.cc"
Dim counter = 103
Dim today = DateTime.Today

// Using data types:
Dim greeting As String = "Welcome to W3CSchool.cc"
Dim counter As Integer = 103
Dim today As DateTime = DateTime.Today
```

## 数据类型

下面列出了常用的数据类型：

| 类型 | 描述 | 实例 |
|---|---|---|
| integer | 整数（全数字） | 103, 12, 5168 |
| double | 64 位浮点数 | 3.14, 3.4e38 |
| decimal | 十进制数字（高精度） | 1037.196543 |
| boolean | 布尔值 | true, false |
| string | 字符串 | "Hello W3CSchool.cc", "John" |

# 运算符

运算符告诉 ASP.NET 在表达式中执行什么样的命令。

VB 语言支持多种运算符。下面列出了常用的运算符：

| 运算符 | 描述 | 实例 |
|---|---|---|
| = | 给一个变量赋值。 | `i=6` |
| + - * / | 加上一个值或者一个变量。 减去一个值或者一个变量。 乘以一个值或者一个变量。 除以一个值或者一个变量。 | `i=5+5` `i=5-5` `i=5*5` `i=5/5` |
| += -= | 变量递增。 变量递减。 | `i += 1` `i -= 1` |
| = | 相等。如果值相等则返回 true。 | `if i=10` |
| <> | 不等。如果值不等则返回 true。 | `if <>10` |
| < > <= >= | 小于。 大于。 小于等于。 大于等于。 | `if i<10` `if i>10` `if i<=10` `if i>=10` |
| & | 连接字符串（一系列互相关联的事物）。 | `"w3" & "schools"` |
| . | 点号。分隔对象和方法。 | `DateTime.Hour` |
| () | 圆括号。将值进行分组。 | `(i+5)` |
| () | 圆括号。传递参数。 | `x=Add(i,5)` |
| () | 圆括号。访问数组或者集合的值。 | `name(3)` |
| Not | 非。真/假取反。 | `if Not ready` |
| And OR | 逻辑与。 逻辑或。 | `if ready And clear` `if ready Or clear` |
| AndAlso orElse | 扩展的逻辑与。 扩展的逻辑或。 | `if ready AndAlso clear` `if ready OrElse clear` |

# 转换数据类型

从一种数据类型转换到另一种数据类型，有时候是很有用的。

最常见的例子是将字符串输入转换为另一种类型，如整数或者日期。

一般规则下，都是将用户输入看做字符串处理，即使用户输入了数字。因此数值输入必须被转换成数字，然后才能将其用于计算。

下面列出了常用的转换方法：

| 方法 | 描述 | 实例 |
| --- | --- | --- |
| AsInt() IsInt() | 转换字符串为整数。 | if myString.IsInt() then myInt=myString.AsInt() end if |
| AsFloat() IsFloat() | 转换字符串为浮点数。 | if myString.IsFloat() then myFloat=myString.AsFloat() end if |
| AsDecimal() IsDecimal() | 转换字符串为十进制数。 | if myString.IsDecimal() then myDec=myString.AsDecimal() end if |
| AsDateTime() IsDateTime() | 转换字符串为 ASP.NET DateTime 类型。 | myString="10/10/2012" myDate=myString.AsDateTime() |
| AsBool() IsBool() | 转换字符串为布尔值。 | myString="True" myBool=myString.AsBool() |
| ToString() | 转换任何数据类型为字符串。 | myInt=1234 myString=myInt.ToString() |

# ASP.NET Razor - VB 循环和数组

语句在循环中会被重复执行。

## For 循环

如果您需要重复执行相同的语句，您可以设定一个循环。

如果您知道要循环的次数，您可以使用 **for** 循环。这种类型的循环在向上计数或向下计数时特别有用：

## 实例

```
<html>
<body>
@For i=10 To 21
@<p>Line #@i</p>
Next i
</body>
</html>
```

运行实例 ?

## For Each 循环

如果您使用的是集合或者数组，您会经常用到 **for each** 循环。

集合是一组相似的对象，for each 循环可以遍历集合直到完成。

下面的实例中，遍历 ASP.NET Request.ServerVariables 集合。

## 实例

```
<html>
<body>
<ul>
@For Each x In Request.ServerVariables
@<li>@x</li>
Next x
</ul>
</body>
</html>
```

运行实例？

# While 循环

**while** 循环是一个通用的循环。

while 循环以 while 关键字开始，后面紧跟着括号，您可以在括号里规定循环将持续多久，然后是重复执行的代码块。

while 循环通常会设定一个递增或者递减的变量用来计数。

下面的实例中，+= 运算符在每执行一次循环时给变量 i 的值加 1。

# 实例

```
<html>
<body>
@Code
Dim i=0
Do While i<5
i += 1
@<p>Line #@i</p>
Loop
End Code
</body>
</html>
```

运行实例？

# 数组

当您要存储多个相似变量但又不想为每个变量都创建一个独立的变量时，可以使用数组来存储：

# 实例

```
@Code
Dim members As String()={"Jani","Hege","Kai","Jim"}
i=Array.IndexOf(members,"Kai")+1
len=members.Length
x=members(2-1)
end Code
<html>
<body>
<h3>Members</h3>
@For Each person In members
@<p>@person</p>
Next person
<p>The number of names in Members are @len</p>
<p>The person at position 2 is @x</p>
<p>Kai is now in position @i</p>
</body>
</html>
```

运行实例？

# ASP.NET Razor - VB 逻辑条件

编程逻辑：根据条件执行代码。

## If 条件

VB 允许根据条件执行代码。

使用 **if** 语句来判断条件。根据判断结果，if 语句返回 true 或者 false：

- if 语句开始一个代码块
- 条件写在 if 和 then 之间
- 如果条件为真，if ... then 和 end if 之间的代码被执行

### 实例

```
@Code
Dim price=50
End Code
<html>
<body>
@If price>30 Then
@<p>The price is too high.</p>
End If
</body>
</html>
```

运行实例？

## Else 条件

if 语句可以包含 **else** 条件。

else 条件定义了当条件为假时被执行的代码。

### 实例

```
@Code
Dim price=20
End Code
<html>
<body>
@if price>30 then
@<p>The price is too high.</p>
Else
@<p>The price is OK.</p>
End If
</body>
</htmlV>
```

运行实例？

注释：在上面的实例中，如果第一个条件为真，if 块的代码将会被执行。else 条件覆盖了除 if 条件之外的"其他所有情况"。

# ElseIf 条件

多个条件判断可以使用 **elseif** 条件：

# 实例

```
@Code
Dim price=25
End Code
<html>
<body>
@If price>=30 Then
@<p>The price is high.</p>
ElseIf price>20 And price<30
@<p>The price is OK.</p>
Else
@<p>The price is low.</p>
End If
</body>
</html>
```

运行实例？

在上面的实例中，如果第一个条件为真，if 块的代码将会被执行。

如果第一个条件不为真且第二个条件为真，elseif 块的代码将会被执行。

elseif 条件的数量不受限制。

如果 if 和 elseif 条件都不为真，最后的 else 块（不带条件）覆盖了"其他所有情况"。

# Select 条件

**select** 块可以用来测试一些单独的条件：

# 实例

```
@Code
Dim weekday=DateTime.Now.DayOfWeek
Dim day=weekday.ToString()
Dim message=""
End Code
<html>
<body>
@Select Case day
Case "Monday"
message="This is the first weekday."
Case "Thursday"
message="Only one day before weekend."
Case "Friday"
message="Tomorrow is weekend!"
Case Else
message="Today is " & day
End Select
<p>@message</p>
</body>
</html>
```

运行实例？

"Select Case" 后面紧跟着测试值（day）。每个单独的测试条件都有一个 case 值和任意数量的代码行。如果测试值与 case 值相匹配，相应的代码行被执行。

select 块有一个默认的情况（Case Else），当所有的指定的情况都不匹配时，它覆盖了"其他所有情况"。

# MVC 教程

# ASP.NET MVC 教程

ASP.NET 是一个使用 HTML、CSS、JavaScript 和服务器脚本创建网页和网站的开发框架。

ASP.NET 支持三种不同的开发模式：
Web Pages（Web 页面）、MVC（Model View Controller 模型-视图-控制器）、Web Forms（Web 窗体）。

本教程介绍 **MVC**。

| Web Pages |
| --- |
| MVC |
| Web Forms |

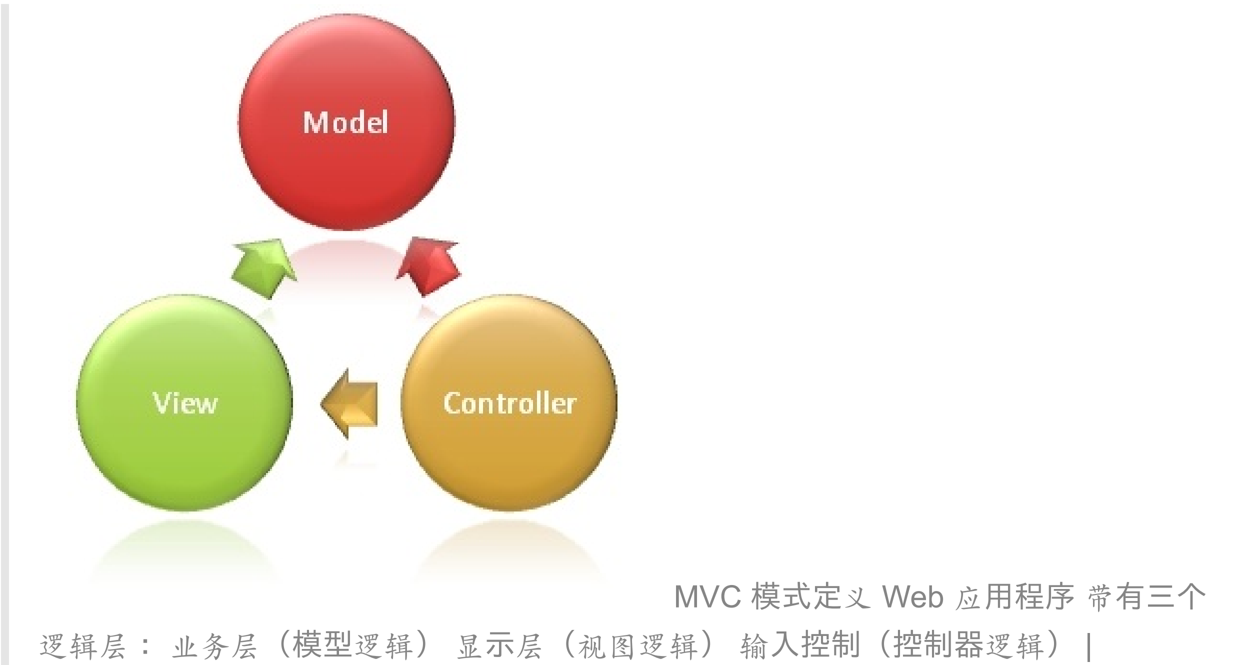
## MVC 编程模式

MVC 是三种 ASP.NET 编程模式中的一种。

MVC 是一种使用 MVC（Model View Controller 模型-视图-控制器）设计创建 Web 应用程序的模式：

- Model（模型）表示应用程序核心（比如数据库记录列表）。
- View（视图）显示数据（数据库记录）。
- Controller（控制器）处理输入（写入数据库记录）。

MVC 模式同时提供了对 HTML、CSS 和 JavaScript 的完全控制。



MVC 模式定义 Web 应用程序 带有三个逻辑层： 业务层（模型逻辑） 显示层（视图逻辑） 输入控制（控制器逻辑） |

**Model**（模型）是应用程序中用于处理应用程序数据逻辑的部分。
通常模型对象负责在数据库中存取数据。

**View**（视图）是应用程序中处理数据显示的部分。
通常视图是依据模型数据创建的。

**Controller**（控制器）是应用程序中处理用户交互的部分。
通常控制器负责从视图读取数据，控制用户输入，并向模型发送数据。

MVC 分层有助于管理复杂的应用程序，因为您可以在一个时间内专门关注一个方面。例如，您可以在不依赖业务逻辑的情况下专注于视图设计。同时也让应用程序的测试更加容易。

MVC 分层同时也简化了分组开发。不同的开发人员可同时开发视图、控制器逻辑和业务逻辑。

# Web Forms 对比 MVC

MVC 编程模式是对传统 ASP.NET（Web Forms）的一种轻量级的替代方案。它是轻量级的、可测试性高的框架，同时整合了所有已有的 ASP.NET 特性，比如母版页、安全性和认证。

# Visual Studio Express 2012/2010

Visual Studio Express 是 Microsoft Visual Studio 的免费版本。

Visual Studio Express 是为 MVC（和 Web Forms）量身定制的开发工具。

Visual Studio Express 包含：

- MVC 和 Web Forms
- 拖拽 Web 控件和 Web 组件
- Web 服务器语言（Razor 使用 VB 或者 C#）
- Web 服务器（IIS Express）
- 数据库服务器（SQL Server Compact）
- 完整的 Web 开发框架（ASP.NET）

如果您已经安装了 Visual Studio Express，您将从本教程中学到更多。

如果您想安装 Visual Studio Express，请点击下列链接中的一个：

Visual Web Developer 2012（Windows 7 或者 Windows 8）

Visual Web Developer 2010（Windows Vista 或者 XP）

>  在您首次安装完 Visual Studio Express 之后，您可以通过再次运行安装程序来安装补丁和服务包，只需要再次点击链接即可。 |

# ASP.NET MVC 参考手册

在本教程的最后，我们提供了完整的 ASP.NET MVC 参考手册供您查阅。

# ASP.NET MVC - Internet 应用程序

为了学习 ASP.NET MVC，我们将构建一个 Internet 应用程序。

第 1 部分：创建应用程序。

## 我们将构建什么

我们将构建一个支持添加、编辑、删除和列出数据库存储信息的 Internet 应用程序。

## 我们将做什么

Visual Web Developer 提供了构建 Web 应用程序的不同模板。

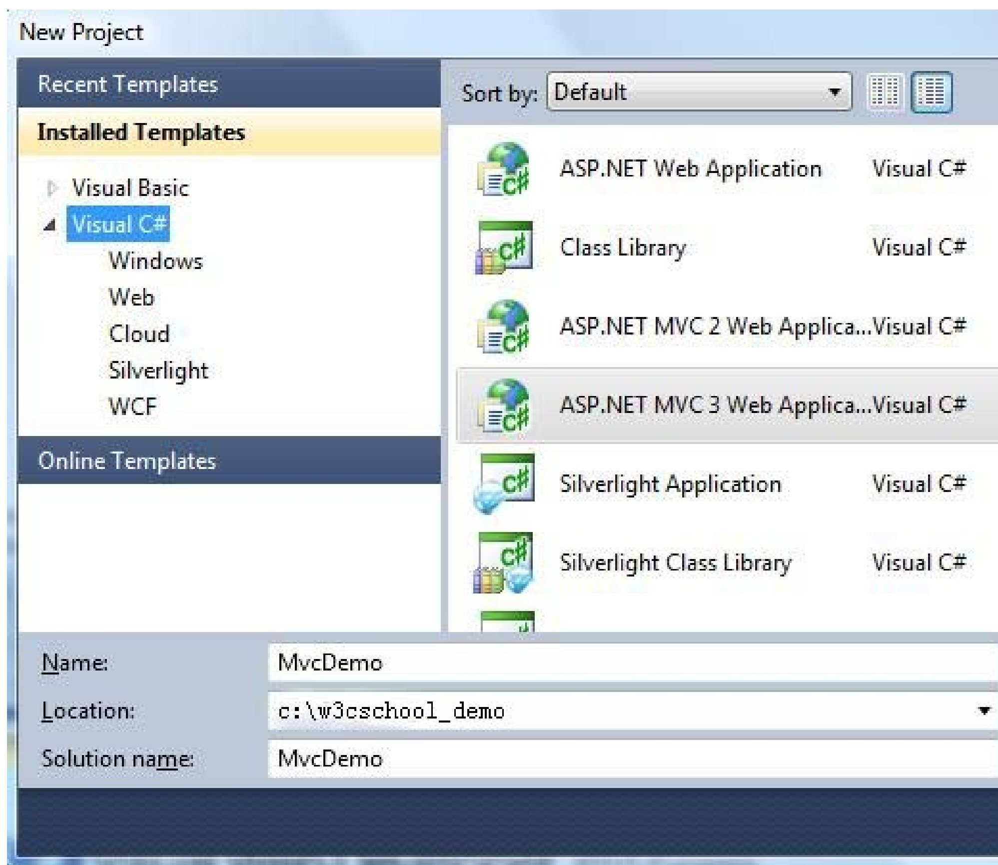
我们将使用 **Visual Web Developer** 来创建一个带 **HTML5** 标记的空的 MVC Internet 应用程序。

当这个空白的 Internet 应用程序被创建之后，我们将逐步向该应用添加代码，直到全部完成。我们将使用 **C#** 作为编程语言，并使用最新的 **Razor** 服务器代码标记。

沿着这个思路，我们将讲解这个应用程序的内容、代码和所有组件。

## 创建 **Web** 应用程序

如果您已经安装了 Visual Web Developer，请启动 Visual Web Developer 并选择 **New Project** 来新建项目。否则您就只能通过阅读教程来学习了。



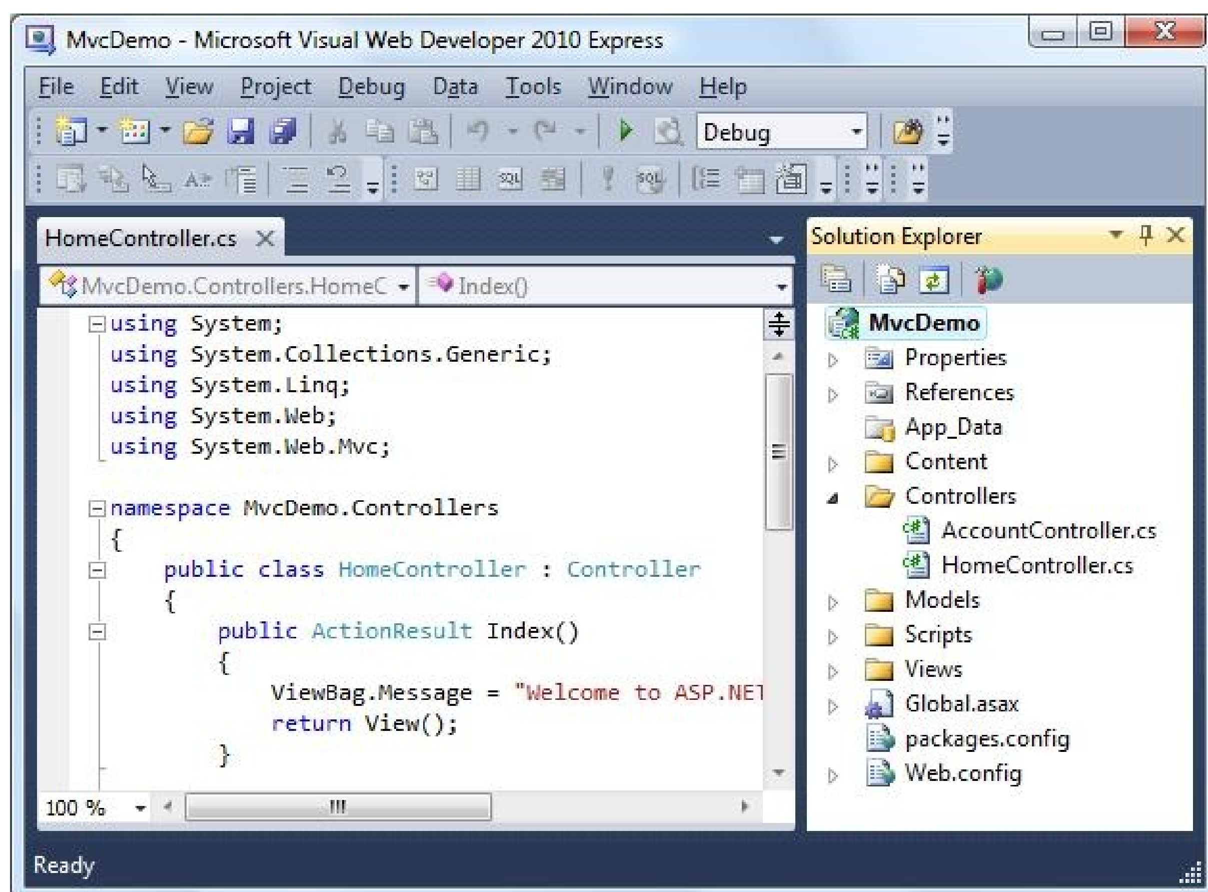

在 New Project 对话框中：

- 打开**Visual C#**模板
- 选择模板 **ASP.NET MVC 3 Web Application**
- 设置项目名称为 **MvcDemo**
- 设置磁盘位置，比如 **c:\w3cschool_demo**
- 点击 **OK**

当 New Project 对话框打开时：

- 选择 **Internet Application** 模板
- 选择 **Razor Engine**（Razor 引擎）
- 选择 **HTML5 Markup**（HTML5 标记）
- 点击 **OK**

Visual Studio Express 将创建一个如下所示的类似项目：




我们将在本教程的下一章中探究有关文件和文件夹的内容。

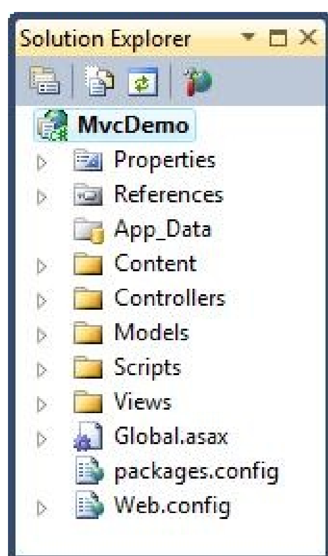
# ASP.NET MVC - 应用程序文件夹

为了学习 ASP.NET MVC，我们将构建一个 Internet 应用程序。

第 2 部分：探究应用程序文件夹。

## MVC 文件夹

一个典型的 ASP.NET MVC Web 应用程序的文件夹内容如下所示：




应用程序信息

Properties
References

应用程序文件夹

App_Data 文件夹
Content 文件夹
Controllers 文件夹
Models 文件夹
Scripts 文件夹
Views 文件夹

配置文件

Global.asax
packages.config
Web.config

所有的 MVC 应用程序的文件夹名称都是相同的。MVC 框架是基于默认的命名。控制器写在 Controllers 文件夹中，视图写在 Views 文件夹中，模型写在 Models 文件夹中。您不必再应用程序代码中使用文件夹名称。

标准化的命名减少了代码量，同时有利于开发人员对 MVC 项目的理解。

下面是对每个文件夹内容的简短概述：

# App_Data 文件夹

**App_Data** 文件夹用于存储应用程序数据。

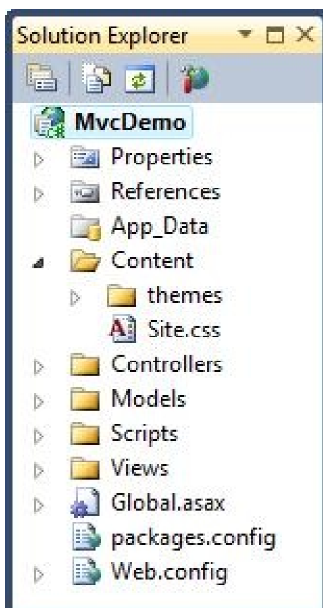
我们将在本教程后面的章节中介绍添加 SQL 数据库到 App_Data 文件夹。

# Content 文件夹

**Content** 文件夹用于存放静态文件，比如样式表（CSS 文件）、图标和图像。

Visual Web Developer 会自动添加一个 **themes** 文件夹到 Content 文件夹中。themes 文件夹存放 jQuery 样式和图片。在项目中，您可以删除这个 themes 文件夹。

Visual Web Developer 同时也会添加一个标准的样式表文件到项目中：即 content 文件夹中的 **Site.css** 文件。这个样式表文件是您想要改变应用程序样式时需要编辑的文件。



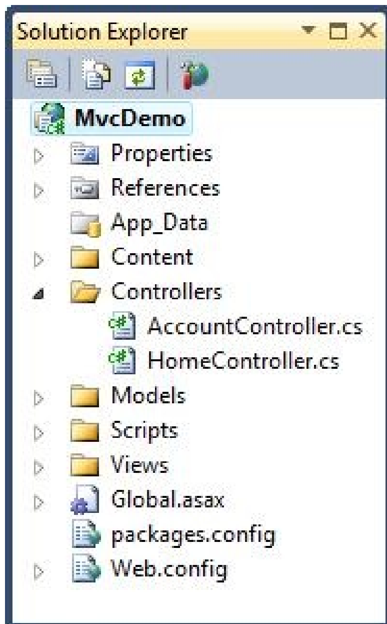

我们将在本教程的下一章中编辑这个样式表文件（Site.css）。

# Controllers 文件夹

Controllers 文件夹包含负责处理用户输入和相应的控制器类。

MVC 要求所有控制器文件的名称以 "Controller" 结尾。

Visual Web Developer 已经创建好一个 Home 控制器（用于 Home 页面和 About 页面）和一个 Account 控制器（用于 Login 页面）：




我们将在本教程后面的章节中创建更多的控制器。

## Models 文件夹

Models 文件夹包含表示应用程序模型的类。模型控制并操作应用程序的数据。

我们将在本教程后面的章节中创建模型（类）。

## Views 文件夹

Views 文件夹用于存储与应用程序的显示相关的 HTML 文件（用户界面）。

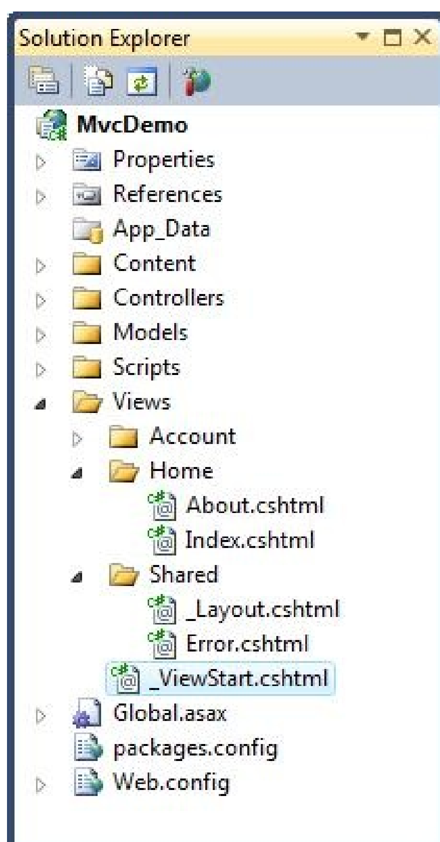
Views 文件夹中包含每个控制器对应的一个文件夹。

在 Views 文件夹中，Visual Web Developer 已经创建了一个 Account 文件夹、一个 Home 文件夹、一个 Shared 文件夹。

Account 文件夹包含用于用户账号注册和登录的页面。

Home 文件夹用于存储诸如 home 页和 about 页之类的应用程序页面。

Shared 文件夹用于存储控制器间分享的视图（母版页和布局页）。

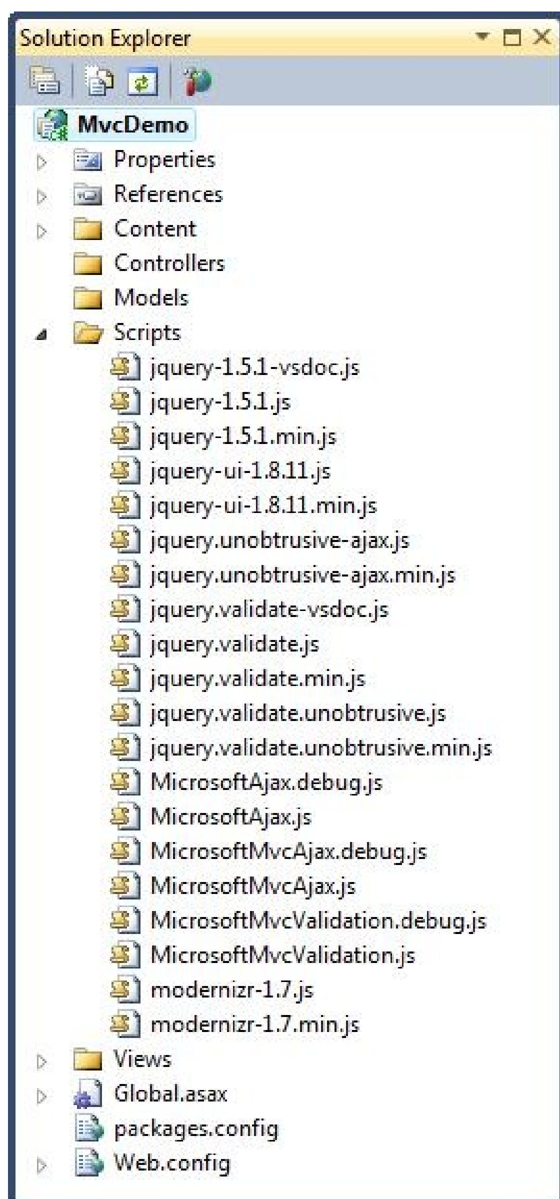



我们将在本教程的下一章中编辑这些布局文件。

## Scripts 文件夹

Scripts 文件夹存储应用程序的 JavaScript 文件。

默认情况下，Visual Web Developer 在这个文件夹中存放标准的 MVC、Ajax 和 jQuery 文件：




注释：名为 "modernizr" 的文件时用于在应用程序中支持 HTML5 和 CSS3 的 JavaScript 文件。

# ASP.NET MVC - 样式和布局

为了学习 ASP.NET MVC，我们将构建一个 Internet 应用程序。

第 3 部分：添加样式和统一的外观（布局）。

## 添加布局

文件 _Layout.cshtml 表示应用程序中每个页面的布局。它位于 Views 文件夹中的 Shared 文件夹。

打开文件 _Layout.cshtml，把内容替换成：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>@ViewBag.Title</title>
<link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />
<script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")"></script>
<script src="@Url.Content("~/Scripts/modernizr-1.7.min.js")"></script>
</head>
<body>
<ul id="menu">
<li>@Html.ActionLink("Home", "Index", "Home")</li>
<li>@Html.ActionLink("Movies", "Index", "Movies")</li>
<li>@Html.ActionLink("About", "About", "Home")</li>
</ul>
<section id="main">
@RenderBody()
<p>Copyright W3CSchool 2012\. All Rights Reserved.</p>
</section>
</body>
</html>
```

## HTML 帮助器

在上面的代码中，HTML 帮助器用于修改 HTML 输出：

```
@Url.Content() - URL 内容将在此处插入。

@Html.ActionLink() - HTML 链接将在此处插入。
```

在本教程后面的章节中，您将学到更多关于 HTML 帮助器的知识。

## Razor 语法

在上面的代码中，红色标记的代码是使用 Razor 标记的 C#。

```
@ViewBag.Title - 页面标题将在此处插入。

@RenderBody() - 页面内容将在此处呈现。
```

您可以在我们的 Razor 教程中学习关于 C# 和 VB（Visual Basic）的 Razor 标记的知识。

# 添加样式

应用程序的样式表是 Site.css，位于 Content 文件夹中。

打开文件 Site.css，把内容替换成：

```
body
{
font: "Trebuchet MS", Verdana, sans-serif;
background-color: #5c87b2;
color: #696969;
}
h1
{
border-bottom: 3px solid #cc9900;
font: Georgia, serif;
color: #996600;
}
#main
{
padding: 20px;
background-color: #ffffff;
border-radius: 0 4px 4px 4px;
}
a
{
color: #034af3;
}
/* Menu Styles ------------------------------*/
ul#menu
{
padding: 0px;
position: relative;
margin: 0;
}
ul#menu li
{
display: inline;
}
ul#menu li a
{
background-color: #e8eef4;
padding: 10px 20px;
text-decoration: none;
line-height: 2.8em;
/*CSS3 properties*/
border-radius: 4px 4px 0 0;
}
ul#menu li a:hover
{
background-color: #ffffff;
}
/* Forms Styles ------------------------------*/
fieldset
{
padding-left: 12px;
}
```

```
fieldset label
{
display: block;
padding: 4px;
}
input[type="text"], input[type="password"]
{
width: 300px;
}
input[type="submit"]
{
padding: 4px;
}
/* Data Styles ------------------------------*/
table.data
{
background-color:#ffffff;
border:1px solid #c3c3c3;
border-collapse:collapse;
width:100%;
}
table.data th
{
background-color:#e8eef4;
border:1px solid #c3c3c3;
padding:3px;
}
table.data td
{
border:1px solid #c3c3c3;
padding:3px;
}
```

# _ViewStart 文件

Shared 文件夹（位于 Views 文件夹内）中的 _ViewStart 文件包含如下内容：

```
@{Layout = "~/Views/Shared/_Layout.cshtml";}
```

这段代码被自动添加到由应用程序显示的所有视图。

如果您删除了这个文件，则必须向所有视图中添加这行代码。

在本教程后面的章节中，您将学到更多关于视图的知识。

# ASP.NET MVC - 控制器

为了学习 ASP.NET MVC，我们将构建一个 Internet 应用程序。

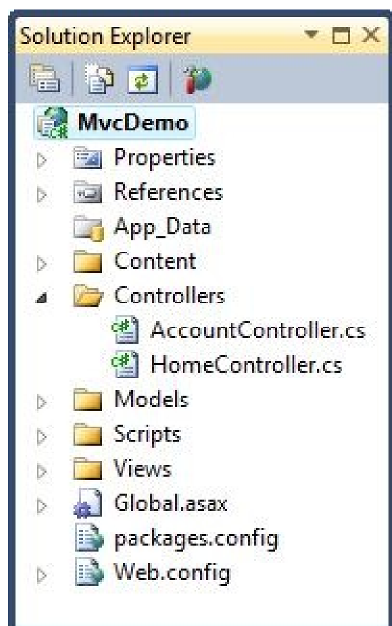
第 4 部分：添加控制器。

## Controllers 文件夹

**Controllers** 文件夹包含负责处理用户输入和响应的控制类。

MVC 要求所有控制器文件的名称以 "Controller" 结尾。

在我们的实例中，Visual Web Developer 已经创建好了一下文件： **HomeController.cs**（用于 Home 页面和 About 页面）和**AccountController.cs** （用于登录页面）：




Web 服务器通常会将进入的 URL 请求直接映射到服务器上的磁盘文件。例如：URL 请求 "http://www.w3cschool.cc/index.php" 将直接映射到服务器根目录上的文件 "index.php"。

MVC 框架的映射方式有所不同。MVC 将 URL 映射到方法。这些方法在类中被称为"控制器"。

控制器负责处理进入的请求，处理输入，保存数据，并把响应发送回客户端。

## Home 控制器

在我们应用程序中的控制器文件**HomeController.cs**，定义了两个控件 **Index** 和 **About**。

把 HomeController.cs 文件的内容替换成：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MvcDemo.Controllers
{
public class HomeController : Controller
{
public ActionResult Index()
{return View();}

public ActionResult About()
{return View();}
}
}
```

# Controller 视图

Views 文件夹中的文件 **Index.cshtml** 和 **About.cshtml** 定义了控制器中的 ActionResult 视图 Index() 和 About()。

# ASP.NET MVC - 视图

为了学习 ASP.NET MVC，我们将构建一个 Internet 应用程序。

第 5 部分：添加用于显示应用程序的视图。

## Views 文件夹

**Views** 文件夹存储的是与应用程序显示（用户界面）相关的文件（HTML 文件）。根据所采用的语言内容，这些文件可能扩展名可能是 html、asp、aspx、cshtml 和 vbhtml。

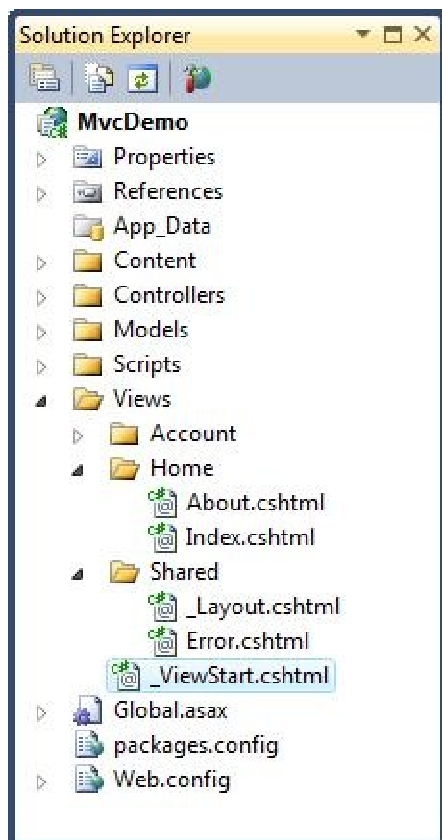
Views 文件夹中包含每个控制器对应的一个文件夹。

在 Views 文件夹中，Visual Web Developer 已经创建了一个 Account 文件夹、一个 Home 文件夹、一个 Shared 文件夹。

Account 文件夹包含用于用户账号注册和登录的页面。

Home 文件夹用于存储诸如 home 页和 about 页之类的应用程序页面。

Shared 文件夹用于存储控制器间分享的视图（母版页和布局页）。



# ASP.NET 文件类型

在 Views 文件夹中可以看到以下 HTML 文件类型：

| 文件类型 | 扩展名 |
|---|---|
| 纯 HTML | .htm or .html |
| 经典 ASP | .asp |
| 经典 ASP.NET | .aspx |
| ASP.NET Razor C# | .cshtml |
| ASP.NET Razor VB | .vbhtml |

# Index 文件

文件 Index.cshtml 表示应用程序的 Home 页面。它是应用程序的默认文件（首页文件）。

在文件中写入以下内容：

```
@{ViewBag.Title = "Home Page";}
<h1>Welcome to W3CSchool.cc</h1>
<p>Put Home Page content here</p>
```

# About 文件

文件 About.cshtml 表示应用程序的 About 页面。

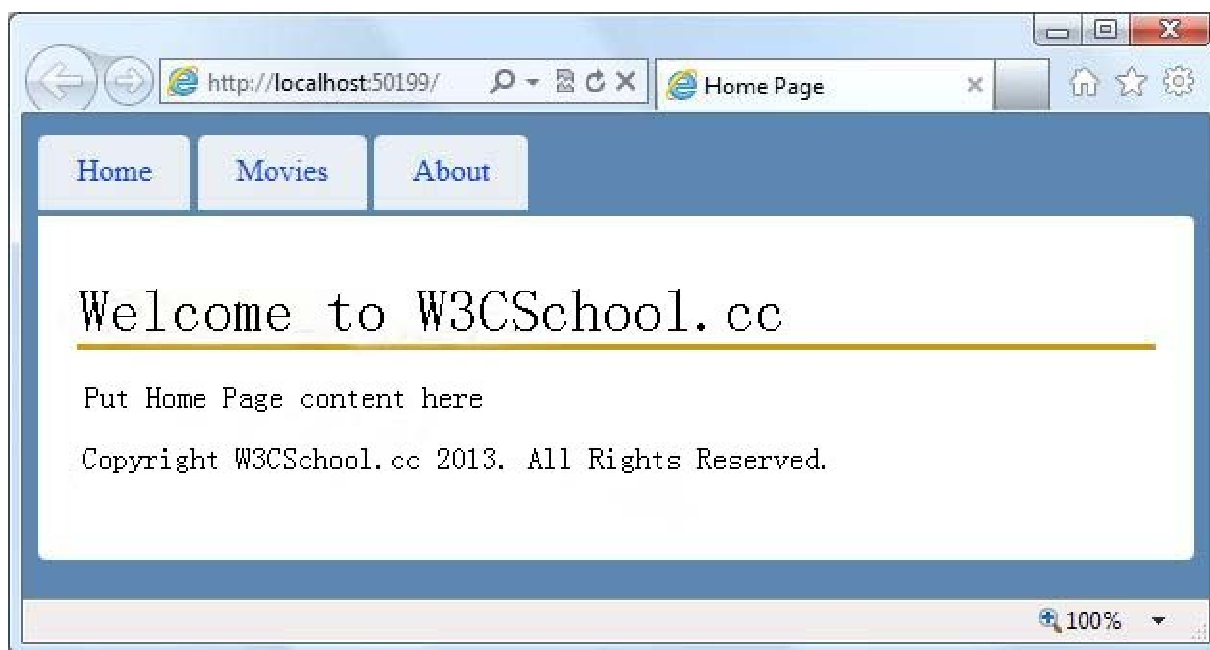
在文件中写入以下内容：

```
@{ViewBag.Title = "About Us";}
<h1>About Us</h1>
<p>Put About Us content here</p>
```

# 运行应用程序

选择 Debug，从 Visual Web Developer 菜单中启动调试 Start Debugging（或者按 F5）。

您的应用程序将显示如下：




点击 "Home" 标签页和 "About" 标签页，看看它是如何运作的。

# 祝贺您

祝贺您。您已经创建好了您的第一个 MVC 应用程序。

注释：您暂时还不能点击 "Movies" 标签页。我们将在本教程的后面章节中为 "Movies" 标签页添加代码。

# ASP.NET MVC - SQL 数据库

为了学习 ASP.NET MVC，我们将构建一个 Internet 应用程序。

第 6 部分：添加数据库。

## 创建数据库

Visual Web Developer 带有名为 SQL Server Compact 免费的 SQL 数据库。

本教程所需的这个数据库可以通过以下几个简单的步骤来创建：

- 右击 **Solution Explorer** 窗口中的 **App_Data** 文件夹
- 选择 **Add, New Item**
- 选择 **SQL Server Compact Local Database** *
- 将数据库命名为 **Movies.sdf**
- 点击 **Add** 按钮

* 如果选项中没有 SQL Server Compact Local Database，则说明您尚未在计算机上安装 SQL Server Compac。请通过以下链接进行安装：SQL Server Compact

Visual Web Developer 会自动在 App_Data 文件夹中创建该数据库。

注释：在本教程中，需要您掌握一些关于 SQL 数据库的基础知识。如果您想先学习这个主题，请访问我们的 SQL 教程。

## 添加数据库表

双击 **App_Data** 文件夹中的 **Movies.sdf** 文件，将打开 **Database Explorer** 窗口。

如需在数据库中创建一个新的表，请右击 **Tables** 文件夹，然后选择 **Create Table**。

创建如下的列：

| 列 | 类型 | 是否允许为 **Null** |
|---|---|---|
| ID | int (primary key) | No |
| Title | nvarchar(100) | No |
| Director | nvarchar(100) | No |
| Date | datetime | No |

对列的解释：

**ID** 是用于标识表中每条记录的整数（全数字）。

**Title** 是 100 个字符长度的文本列，用于存储影片的名称。

**Director** 是 100 个字符长度的文本列，用于存储导演的名字。

**Date** 是日期列，用于存储影片的发布日期。

在创建好上述列之后，您必须将 ID 列设置为表的主键（记录标识符）。要做到这点，请点击列名（ID），并选择 **Primary Key**。在 **Column Properties** 窗口中，设置 **Identity** 属性为 **True**：

Name:

| Column Name | Data Type | Len... | Allow Nulls | Unique | Primary Key |
|---|---|---|---|---|---|
| ID | int | 4 | No | Yes | Yes |
| Title | nvarchar | 100 | No | No | No |
| Director | nvarchar | 100 | No | No | No |
| Date | datetime | 8 | No | No | No |

Delete

| | |
|---|---|
| Default Value | |
| Identity | True |
| Identity Increment | 1 |
| Identity Seed | 1 |
| Is RowGuid | False |
| Precision | |
| Scale | |

当您创建好表列后，保存表并命名为 **MovieDBs**。

注释：

我们特意把表命名为 "MovieDBs"（以 s 结尾）。在下一章中，您将看到用于数据模型的 "MovieDB"。这看起来有点奇怪，不过这种命名惯例能确保控制器连接上数据库表，您必须这么使用。

# 添加数据库记录

您可以使用 Visual Web Developer 向 movie 数据库中添加一些测试记录。

双击 **App_Data** 文件夹中的 **Movies.sdf** 文件。

右击 Database Explorer 窗口中的 **MovieDBs** 表，并选择 **Show Table Data**。

添加一些记录：

| ID | Title | Director | Date |
|---|---|---|---|
| 1 | Psycho | Alfred Hitchcock | 01.01.1960 |
| 2 | La Dolce Vita | Federico Fellini | 01.01.1960 |

注释：ID 列会自动更新，您可以不用编辑它。

# 添加连接字符串

向您的 **Web.config** 文件中的 **<connectionStrings>** 元素添加如下元素：

# ASP.NET MVC - 模型

为了学习 ASP.NET MVC，我们将构建一个 Internet 应用程序。

第 7 部分：添加数据模型。

## MVC 模型

MVC 模型包含了除纯视图和控制器逻辑以外的其他所有应用程序逻辑（业务逻辑、验证逻辑、数据访问逻辑）。

通过 MVC，模型可以控制并操作应用程序数据。

## Models 文件夹

**Models** 文件夹包含表示应用程序模型的类。

Visual Web Developer 自动创建一个 **AccountModels.cs** 文件，该文件包含用于应用程序安全的模型。

**AccountModels** 包含 **LogOnModel**、**ChangePasswordModel** 和 **RegisterModel**。

## 添加数据库模型

本教程所需的数据库模型可以通过以下几个简单的步骤来创建：

- 在 **Solution Explorer**窗口中，右击 **Models** 文件夹，并选择 **Add** 和 **Class**。
- 将类命名为 **MovieDB.cs**，然后点击 **Add**。
- 编辑这个类：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;

namespace MvcDemo.Models
{
public class MovieDB
{
public int ID { get; set; }
public string Title { get; set; }
public string Director { get; set; }
public DateTime Date { get; set; }

}
public class MovieDBContext : DbContext
{
public DbSet<MovieDB> Movies { get; set; }
}
}
```

注释：

我们特意把模型命名为 "MovieDB"。在上一章中，您已经看到用于数据库表的 "MovieDBs"（以 s 结尾）。这看起来有点奇怪，不过这种命名惯例能确保模型连接上数据库表，您必须这么使用。

# 添加数据库控制器

本教程所需的数据库控制器可以通过以下几个简单的步骤来创建：

- 重建您的项目：选择 **Debug**，然后从菜单中选择 **Build MvcDemo**。
- 在 Solution Explorer（解决方案资源管理器）中，右击 **Controllers** 文件夹，选择 **Add** 和 **Controller**。
- 设置控制器名称为 **MoviesController**。
- 选择模板：**Controller with read/write actions and views, using Entity Framework**
- 选择模型类：**MovieDB (MvcDemo.Models)**
- 选择 data context 类：**MovieDBContext (MvcDemo.Models)**
- 选择视图 **Razor (CSHTML)**
- 点击 **Add**

Visual Web Developer 将创建以下文件：

- **Controllers** 文件夹中的 **MoviesController.cs** 文件
- **Views** 文件夹中的 **Movies** 文件夹

# 添加数据库视图

在 Movies 文件夹中，会自动创建以下文件：

- Create.cshtml
- Delete.cshtml
- Details.cshtml
- Edit.cshtml
- Index.cshtml

# 祝贺您

祝贺您。您已经向应用程序添加了您的第一个 MVC 数据模型。

现在您可以点击 "Movies" 标签页了。

# ASP.NET MVC - 安全

为了学习 ASP.NET MVC，我们将构建一个 Internet 应用程序。

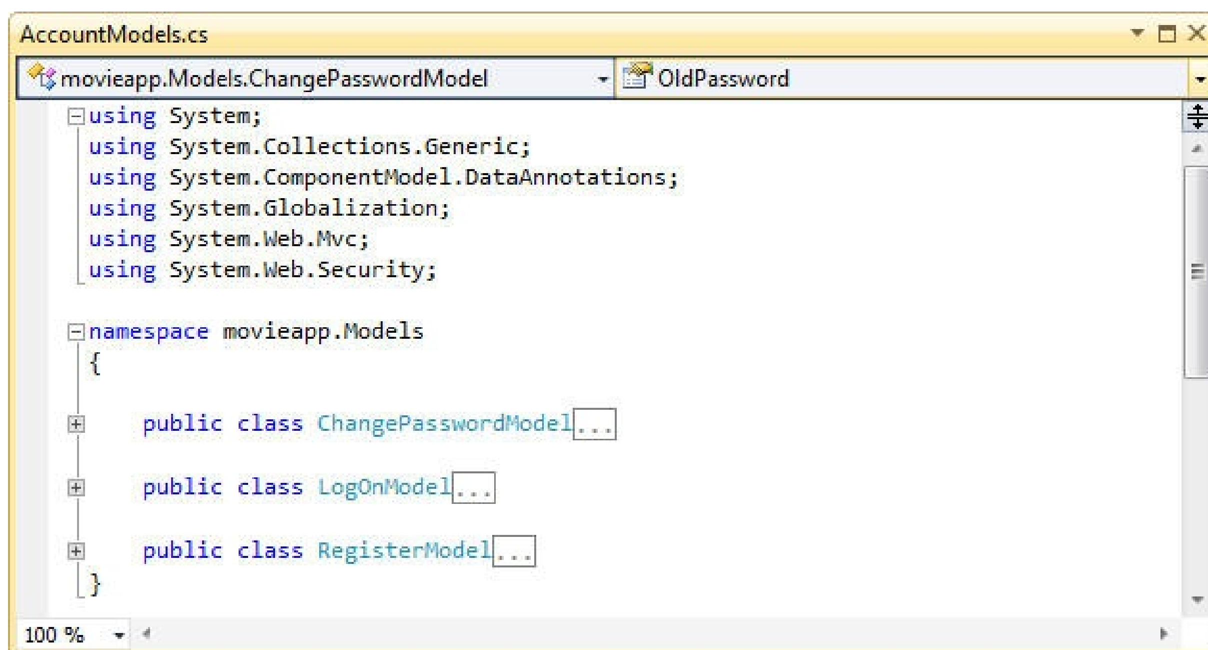
第 8 部分：添加安全。

## MVC 应用程序安全

**Models** 文件夹包含表示应用程序模型的类。

Visual Web Developer 自动创建 **AccountModels.cs** 文件，该文件包含用于应用程序认证的模型。

**AccountModels** 包含 **LogOnModel**、**ChangePasswordModel** 和 **RegisterModel**：




## Change Password 模型

```
public class ChangePasswordModel
{

[Required]
[DataType(DataType.Password)]
[Display(Name = "Current password")]
public string OldPassword { get; set; }

[Required]
[StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.", Minimu
[DataType(DataType.Password)]
[Display(Name = "New password")]
public string NewPassword { get; set; }

[DataType(DataType.Password)]
[Display(Name = "Confirm new password")]
[Compare("NewPassword", ErrorMessage = "The new password and confirmation password do not
public string ConfirmPassword { get; set; }

}
```

# Logon 模型

```
public class LogOnModel
{

[Required]
[Display(Name = "User name")]
public string UserName { get; set; }

[Required]
[DataType(DataType.Password)]
[Display(Name = "Password")]
public string Password { get; set; }

[Display(Name = "Remember me?")]
public bool RememberMe { get; set; }

}
```

# Register 模型

```
public class RegisterModel
{

[Required]
[Display(Name = "User name")]
public string UserName { get; set; }

[Required]
[DataType(DataType.EmailAddress)]
[Display(Name = "Email address")]
public string Email { get; set; }

[Required]
[StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.", Minimu
[DataType(DataType.Password)]
[Display(Name = "Password")]
public string Password { get; set; }

[DataType(DataType.Password)]
[Display(Name = "Confirm password")]
[Compare("Password", ErrorMessage = "The password and confirmation password do not match.
public string ConfirmPassword { get; set; }

}
```

# ASP.NET MVC - HTML 帮助器

HTML 帮助器用于修改 HTML 输出。

## HTML 帮助器

通过 MVC，HTML 帮助器类似于传统的 ASP.NET Web Form 控件。

就像 ASP.NET 中的 Web Form 控件，HTML 帮助器用于修改 HTML。但是 HTML 帮助器是更轻量级的。与 Web Form 控件不同，HTML 帮助器没有事件模型和视图状态。

在大多数情况下，HTML 帮助器仅仅是一个返回字符串的方法。

通过 MVC，您可以创建您自己的帮助器，或者直接使用内建的 HTML 帮助器。

## 标准的 HTML 帮助器

MVC 包含了大多数常用的 HTML 元素类型的标准帮助器，比如 HTML 链接和 HTML 表单元素。

## HTML 链接

呈现 HTML 链接的最简单的方法是使用 HTML.ActionLink() 帮助器。

通过 MVC，Html.ActionLink() 不连接到视图。它创建一个连接到控制器操作。

Razor 语法：

```
@Html.ActionLink("About this Website", "About")
```

ASP 语法：

```
<%=Html.ActionLink("About this Website", "About")%>
```

第一个参数是链接文本，第二个参数是控制器操作的名称。

上面的 Html.ActionLink() 帮助器，输出以下的 HTML：

```
<a href="/Home/About">About this Website</a>
```

Html.ActionLink() 帮助器的一些属性：

| 属性 | 描述 |
|---|---|
| .linkText | URL 文本（标签），定位点元素的内部文本。 |
| .actionName | 操作（action）的名称。 |
| .routeValues | 传递给操作（action)的值，是一个包含路由参数的对象。 |
| .controllerName | 控制器的名称。 |
| .htmlAttributes | URL 的属性设置，是一个包含要为该元素设置的 HTML 特性的对象。 |
| .protocol | URL 协议，如 "http" 或 "https"。 |
| .hostname | URL 的主机名。 |
| .fragment | URL 片段名称（定位点名称）。 |

注释：您可以向控制器操作传递值。例如，您可以向数据库 Edit 操作传递数据库记录的 id：

Razor 语法 C#：

```
@Html.ActionLink("Edit Record", "Edit", new {Id=3})
```

Razor 语法 VB：

```
@Html.ActionLink("Edit Record", "Edit", New With{.Id=3})
```

上面的 Html.ActionLink() 帮助器，输出以下的 HTML：

```
<a href="/Home/Edit/3">Edit Record</a>
```

# HTML 表单元素

以下 HTML 帮助器可用于呈现（修改和输出）HTML 表单元素：

- BeginForm()
- EndForm()
- TextArea()
- TextBox()
- CheckBox()
- RadioButton()
- ListBox()
- DropDownList()

- Hidden()
- Password()

ASP.NET 语法 C#：

```
<%= Html.ValidationSummary("Create was unsuccessful. Please correct the errors and try ag
<% using (Html.BeginForm()){%>
<p>
<label for="FirstName">First Name:</label>
<%= Html.TextBox("FirstName") %>
<%= Html.ValidationMessage("FirstName", "*") %>
</p>
<p>
<label for="LastName">Last Name:</label>
<%= Html.TextBox("LastName") %>
<%= Html.ValidationMessage("LastName", "*") %>
</p>
<p>
<label for="Password">Password:</label>
<%= Html.Password("Password") %>
<%= Html.ValidationMessage("Password", "*") %>
</p>
<p>
<label for="Password">Confirm Password:</label>
<%= Html.Password("ConfirmPassword") %>
<%= Html.ValidationMessage("ConfirmPassword", "*") %>
</p>
<p>
<label for="Profile">Profile:</label>
<%= Html.TextArea("Profile", new {cols=60, rows=10})%>
</p>
<p>
<%= Html.CheckBox("ReceiveNewsletter") %>
<label for="ReceiveNewsletter" style="display:inline">Receive Newsletter?</label>
</p>
<p>
<input type="submit" value="Register" />
</p>
<%}%>
```

# ASP.NET MVC - 发布网站

学习如何在不使用 Visual Web Developer 的情况下发布 MVC 应用程序。

## 在不使用 Visual Web Developer 的情况下发布您的应用程序

通过在 WebMatrix、Visual Web Developer 或 Visual Studio 中使用发布命令，可以发布一个 ASP.NET MVC 应用程序到远程服务器上。

此功能会复制所有您的应用程序文件、控制器、模型、图像以及用于 MVC、Web Pages、Razor、Helpers、SQL Server Compact（如果使用数据库）所有必需的 DLL 文件。

有时您不希望使用这些选项。或许您的主机提供商仅支持 FTP？或许您的网站基于经典 ASP？或许您希望亲自拷贝这些文件？又或许您希望使用 Front Page、Expression Web 等其他一些发布软件？

您会遇到问题吗？是的，会的。但是您有办法解决它。

要执行网站复制，您必须知道如何引用正确的文件，哪些 DLL 文件需要复制，并在何处存储它们。

请按照下列步骤操作：

## 1. 使用最新版本的 ASP.NET

在您继续操作之前，请确保您的主机运行的是最新版的 ASP.NET（4.0 或者 4.5）。

## 2. 复制 Web 文件夹

从您的开发计算机上复制您的网站（所有文件夹和内容）到远程主机（服务器）上的应用程序文件夹中。

如果您的 **App_Data** 文件夹中包含测试数据，请不要复制这个 App_Data 文件夹（详见下面的第 5 点）。

## 3. 复制 DLL 文件

在远程服务器上的应用程序根目录中创建 bin 文件夹。（如果您已经安装 Helpers，则 bin 文件夹已经存在）

复制下列文件夹中的所有文件：

**C:Program Files (x86)Microsoft ASP.NETASP.NET Web Pagesv1.0Assemblies**

**C:Program Files (x86)Microsoft ASP.NETASP.NET MVC 3Assemblies**

到您的远程服务器上的应用程序的 bin 文件夹中。

# 4. 复制 SQL Server Compact DLL 文件

如果您的应用程序使用了 SQL Server Compact 数据库（在 App_Data 文件夹中的一个 .sdf 文件），那么您必须复制 SQL Server Compact DLL 文件：

复制下列文件夹中的所有文件：

**C:Program Files (x86)Microsoft SQL Server Compact Editionv4.0Private**

到您的远程服务器上的应用程序的 bin 文件夹中。

创建（或者编辑）应用程序的 Web.config 文件：

## 实例 C\

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
<system.data>
<DbProviderFactories>
<remove invariant="System.Data.SqlServerCe.4.0" />

<add invariant="System.Data.SqlServerCe.4.0"
name="Microsoft SQL Server Compact 4.0"
description=".NET Framework Data Provider for Microsoft SQL Server Compact" type="System.

</DbProviderFactories>
</system.data>
</configuration>
```

# 5. 复制 SQL Server Compact 数据

您的 App_Data 文件夹中有没有包含测试数据的 .sdf 文件？

您是否希望发布您的测试数据到远程服务器上？

大多数时候一般是不希望。

如果您一定要复制 SQL 数据文件（.sdf 文件），那么您应该删除数据库中的所有数据，然后从您的开发计算机上复制一个空的 .sdf 文件到服务器上。

就是这样。**GOOD LUCK**！

# Web Forms 教程

# ASP.NET Web Forms - 教程

ASP.NET 是一个使用 HTML、CSS、JavaScript 和服务器脚本创建网页和网站的开发框架。

ASP.NET 支持三种不同的开发模式：
Web Pages（Web 页面）、MVC（Model View Controller 模型-视图-控制器）、Web Forms（Web 窗体）：

本教程介绍 **Web Forms**。

| Web Pages |
| --- |
| MVC |
| Web Forms |

## 从何入手？

多数开发人员学习一个新技术，是从查看运行实例开始的。

如果您想查看一个 Web Forms 运行实例，请查看以下的 ASP.NET Web Forms 演示。

## 什么是 Web Forms？

Web Forms 是三种创建 ASP.NET 网站和 Web 应用程序的编程模式中的一种。

其他两种编程模式是 Web Pages 和 MVC（Model View Controller 模型-视图-控制器）。

Web Forms 是最古老的 ASP.NET 编程模式，是整合了 HTML、服务器控件和服务器代码的事件驱动网页。

Web Forms 是在服务器上编译和执行的，再由服务器生成 HTML 显示为网页。

Web Forms 有数以百计的 Web 控件和 Web 组件用来创建带有数据访问的用户驱动网站。

# Visual Studio Express 2012/2010

Visual Studio Express 是 Microsoft Visual Studio 的免费版本。

Visual Studio Express 是为 Web Forms（和 MVC）量身定制的开发工具。

Visual Studio Express 包含：

- MVC 和 Web Forms

- 拖拽 Web 控件和 Web 组件
- Web 服务器语言（Razor 使用 VB 或者 C#）
- Web 服务器（IIS Express）
- 数据库服务器（SQL Server Compact）
- 完整的 Web 开发框架（ASP.NET）

如果您已经安装了 Visual Studio Express，您将从本教程中学到更多。

如果您想安装 Visual Studio Express，请点击下列链接中的一个：

Visual Web Developer 2012（Windows 7 或者 Windows 8）

Visual Web Developer 2010（Windows Vista 或者 XP）

# ASP.NET 参考手册

在本教程的最后，您将看到一套完整的 ASP.NET 参考手册，介绍了对象、组件、属性和方法。

ASP.NET 参考手册

# ASP.NET Web Forms - HTML 页面

简单的 ASP.NET 页面看上去就像普通的 HTML 页面。

## Hello W3CSchool.cc

在开始学习 ASP.NET 之前，我们先来构建一个简单的 HTML 页面，该页面将在浏览器中显示 "Hello W3CSchool.cc"：

```
<table bgcolor="yellow" border="1" width="100%">

<tbody>

<tr><td>

## Hello W3CSchool.cc!

</td></tr>

</tbody>

</table>
```

## 用 HTML 编写的 Hello W3CSchool.cc

下面的代码将以 HTML 页面的形式显示实例：

```
<html>
<body bgcolor="yellow">
<center>
<h2>Hello W3CSchool.cc!</h2>
</center>
</body>
</html>
```

如果您想亲自尝试一下，请保存上面的代码到一个名为 "**firstpage.htm**" 的文件中，并创建一个到该文件的链接：firstpage.htm。

## 用 ASP.NET 编写的 Hello W3CSchool.cc

转换 HTML 页面为 ASP.NET 页面最简单的方法是，直接复制一个 HTML 文件，并把新文件的扩展名改成 **.aspx** 。

下面的代码将以 ASP.NET 页面的形式显示实例：

```
<html>
<body bgcolor="yellow">
<center>
<h2>Hello W3CSchool.cc!</h2>
</center>
</body>
</html>
```

如果您想亲自尝试一下，请保存上面的代码到一个名为 "**firstpage.aspx**" 的文件中，并创建一个到该文件的链接：firstpage.aspx。

# 它是如何工作的？

从根本上讲，ASP.NET 页面与 HTML 是完全相同的。

HTML 页面的扩展名是 .htm。如果浏览器向服务器请求一个 HTML 页面，服务器可以不进行任何修改，就直接发送页面给浏览器。

ASP.NET 页面的扩展名是 .aspx。如果浏览器向服务器请求个 ASP.NET 页面，服务器在将结果发回给浏览器之前，需要先处理页面中的可执行代码。

上面的 ASP.NET 页面不包含任何可执行的代码，所以没有执行任何东西。在下面的实例中，我们将添加一些可执行的代码到页面中，以便演示静态 HTML 页面和动态 ASP 页面的不同之处。

# 经典 ASP

Active Server Pages (ASP) 已经流行很多年了。通过 ASP，可以在 HTML 页面中放置可执行代码。

之前的 ASP 版本（在 ASP.NET 之前）通常被称为经典 ASP。

ASP.NET 不完全兼容经典 ASP，但是只需要经过少量的修改，大部分经典 ASP 页面就可以作为 ASP.NET 页面良好地运行。

如果您想学习更多关于经典 ASP 的知识，请访问我们的 ASP 教程。

# 用经典 ASP 编写的动态页面

为了演示 ASP 是如何显示包含动态内容的页面，我们将向上面的实例中添加一些可执行的代码（红色字体标识）：

```
<html>
<body bgcolor="yellow">
<center>
<h2>Hello W3CSchool.cc!</h2>
<p><%Response.Write(now())%></p>
</center>
</body>
</html>
```

<% --%> 标签内的代码是在服务器上执行的。

Response.Write 是用来向 HTML 输出流中写东西的 ASP 代码。

Now() 是一个返回服务器当前日期和时间的函数。

如果您想亲自尝试一下，请保存上面的代码到一个名为 "**dynpage.asp**" 的文件中，并创建一个到该文件的链接：dynpage.asp。

# 用 ASP .NET 编写的动态页面

下面的代码将以 ASP.NET 页面的形式显示实例：

```
<html>
<body bgcolor="yellow">
<center>
<h2>Hello W3CSchool.cc!</h2>
<p><%Response.Write(now())%></p>
</center>
</body>
</html>
```

如果您想亲自尝试一下，请保存上面的代码到一个名为 "**dynpage.aspx**" 的文件中，并创建一个到该文件的链接：dynpage.aspx。

# ASP.NET 对比经典 ASP

上面的实例无法演示 ASP.NET 与经典 ASP 之间任何的不同之处。

正如最后的两个实例中，您看不出 ASP 页面和 ASP.NET 页面两者之间的不同之处。

在下一章中，您将看到服务器控件是如何让 ASP.NET 比经典 ASP 更强大的。

# ASP.NET Web Forms - 服务器控件

服务器控件是服务器可理解的标签。

## 经典 ASP 的局限性

下面列出的代码是从上一章中复制的：

```
<html>
<body bgcolor="yellow">
<center>
<h2>Hello W3CSchool.cc!</h2>
<p><%Response.Write(now())%></p>
</center>
</body>
</html>
```

上面的代码反映出经典 ASP 的局限性：代码块必须放置在您想要输出显示的位置。

通过经典 ASP，想要把可执行代码从 HTML 页面中分离出来是不可能的。这让页面变得难以阅读，也难以维护。

## ASP.NET - 服务器控件

ASP.NET 通过服务器控件，已经解决了上述的"意大利面条式代码"问题。

服务器控件是服务器可理解的标签。

有三种类型的服务器控件：

- HTML 服务器控件 - 创痛的 HTML 标签
- Web 服务器控件 - 新的 ASP.NET 标签
- Validation 服务器控件 - 用于输入验证

## ASP.NET - HTML 服务器控件

HTML 服务器控件是服务器可理解的 HTML 标签。

ASP.NET 文件中的 HTML 元素，默认是作为文本进行处理的。要想让这些元素可编程，需向 HTML 元素中添加 runat="server" 属性。这个属性表示，该元素将被作为服务器控件进行处理。同时需要添加 id 属性来标识服务器控件。id 引用可用于操作运行时的服务器控件。

注释：所有 HTML 服务器控件必须位于带有 runat="server" 属性的 <form> 标签内。runat="server" 属性表明了该表单必须在服务器上进行处理。同时也表明了包含在它内部的控件可被服务器脚本访问。

在下面的实例中，我们在 .aspx 文件中声明了一个 HtmlAnchor 服务器控件。然后我们在一个事件句柄（事件句柄是一种针对给定事件执行代码的子例程）中操作 HtmlAnchor 控件的 HRef 属性。Page_Load 事件是 ASP.NET 可理解的多种事件中的一种：

```
<script runat="server">
Sub Page_Load
link1.HRef="http://www.w3cschool.cc"
End Sub
</script>

<html>
<body>

<form runat="server">
<a id="link1" runat="server">Visit W3CSchool.cc!</a>
</form>

</body>
</html>
```

可执行代码本身已经被移到 HTML 之外了。

# ASP.NET - Web 服务器控件

Web 服务器控件是服务器可理解的特殊 ASP.NET 标签。

就像 HTML 服务器控件，Web 服务器控件也是在服务器上创建的，它们同样需要 runat="server" 属性才能生效。然而，Web 服务器控件没有必要映射任何已存在的 HTML 元素，它们可以表示更复杂的元素。

创建 Web 服务器控件的语法是：

```
<asp:control_name id="some_id" runat="server" />
```

在下面的实例中，我们在 .aspx 文件中声明了一个 Button 服务器控件。然后我们为 Click 事件创建一个事件句柄，用来改变按钮上的文本：

```
<script runat="server">
Sub submit(Source As Object, e As EventArgs)
button1.Text="You clicked me!"
End Sub
</script>

<html>
<body>

<form runat="server">
<asp:Button id="button1" Text="Click me!"
runat="server" OnClick="submit"/>
</form>

</body>
</html>
```

# ASP.NET - Validation 服务器控件

Validation 服务器控件是用来验证用户输入的。如果用户输入没有通过验证，将显示一条错误消息给用户。

每种 validation 控件执行一种指定类型的验证（比如验证摸个指定的值或者某个范围的值）。

在默认情况下，当 Button、ImageButton、LinkButton 控件被点击时，会执行页面验证。您可以设置 CausesValidation 为 false， 来阻止按钮控件被点击时进行验证。

创建 Validation 服务器控件的语法是：

```
<asp:control_name id="some_id" runat="server" />
```

在下面的实例中，我们在 .aspx 文件中声明了一个 TextBox 控件、一个 Button 控件、一个 RangeValidator 控件。如果验证失败，文本 "The value must be from 1 to 100!" 将会显示在 RangeValidator 控件中：

# 实例

```
<html>
<body>

<form runat="server">
<p>Enter a number from 1 to 100:
<asp:TextBox id="tbox1" runat="server" />
<br /><br />
<asp:Button Text="Submit" runat="server" />
</p>

<p>
<asp:RangeValidator
ControlToValidate="tbox1"
MinimumValue="1"
MaximumValue="100"
Type="Integer"
Text="The value must be from 1 to 100!"
runat="server" />
</p>
</form>

</body>
</html>
```

# ASP.NET Web Forms - 事件

事件句柄是一种针对给定事件来执行代码的子例程。

## ASP.NET - 事件句柄

请看下面的代码：

```
<%
lbl1.Text="The date and time is " & now()
%>

<html>
<body>
<form runat="server">
<h3><asp:label id="lbl1" runat="server" /></h3>
</form>
</body>
</html>
```

上面的代码将在何时被执行？答案是："不知道..."。

## Page_Load 事件

Page_Load 事件是 ASP.NET 可理解的众多事件之一。Page_Load 事件会在页面加载时被触发， ASP.NET 将自动调用 Page_Load 子例程，并执行其中的代码：

## 实例

```
<script runat="server">
Sub Page_Load
lbl1.Text="The date and time is " & now()
End Sub
</script>

<html>
<body>
<form runat="server">
<h3><asp:label id="lbl1" runat="server" /></h3>
</form>
</body>
</html>
```

演示实例？

注释：Page_Load 事件不包含对象引用或事件参数！

# Page.IsPostBack 属性

Page_Load 子例程会在页面每次加载时运行。如果您只想在页面第一次加载时执行 Page_Load 子例程中的代码，那么您可以使用 Page.IsPostBack 属性。如果 Page.IsPostBack 属性设置为 false，则页面第一次被载入，如果设置为 true，则页面被传回到服务器（比如，通过点击表单上的按钮）：

# 实例

```
<script runat="server">
Sub Page_Load
if Not Page.IsPostBack then
lbl1.Text="The date and time is " & now()
end if
End Sub

Sub submit(s As Object, e As EventArgs)
lbl2.Text="Hello World!"
End Sub
</script>

<html>
<body>
<form runat="server">
<h3><asp:label id="lbl1" runat="server" /></h3>
<h3><asp:label id="lbl2" runat="server" /></h3>
<asp:button text="Submit" onclick="submit" runat="server" />
</form>
</body>
</html>
```

演示实例？

上面的实例仅在页面第一次加载时显示 "The date and time is...." 消息。当用户点击 Submit 按钮是，submit 子例程将会在第二个 label 中写入 "Hello World!"，但是第一个 label 中的日期和时间不会改变。

# ASP.NET Web Forms - HTML 表单

所有的服务器控件都必须出现在 <form> 标签中，<form> 标签必须包含 runat="server" 属性。

## ASP.NET Web 表单

所有的服务器控件都必须出现在 <form> 标签中，<form> 标签必须包含 runat="server" 属性。runat="server" 属性表明该表单必须在服务器上进行处理。同时也表明了包含在它内部的控件可被服务器脚本访问：

```
<form runat="server">

...HTML + server controls

</form>
```

注释：该表单总是被提交到自身页面。如果您指定了一个 action 属性，它会被忽略。如果您省略了 metion 属性，它将会默认设置 method="post"。同时，如果您没有指定 name 和 id 属性，它们会由 ASP.NET 自动分配。

注释：一个 .aspx 页面只能包含一个 <form runat="server"> 控件！

如果您在一个包含不带有 name、method、action 或 id 属性的表单的 .aspx 页面中选择查看源代码，您会看到 ASP.NET 添加这些属性到表单上了，如下所示：

```
<form name="_ctl0" method="post" action="page.aspx" id="_ctl0">

...some code

</form>
```

## 提交表单

表单通常通过点击按钮来提交。ASP.NET 中的 Button 服务器控件的格式如下：

```
<asp:Button id="id" text="label" OnClick="sub" runat="server" />
```

id 属性为按钮定义了一个唯一的名称，text 属性为按钮分配了一个标签。onClick 事件句柄规定了一个要执行的已命名的子例程。

在下面的实例中，我们在 .aspx 文件中声明了一个 Button 控件。点击按钮运行改变按钮上文本的子例程：

实例

# ASP.NET Web Forms - 维持 ViewState

通过在您的 Web Form 中维持对象的 ViewState（视图状态），您可以省去大量的编码工作。

## 维持 ViewState（视图状态）

在经典 ASP 中，当一个表单被提交时，所有的表单值都会被清空。假设您提交了一个带有大量信息的表单，而服务器返回了一个错误。您不得不回到表单改正信息。您点击返回按钮，然后发生了什么......所有表单值都被清空了，您不得不重新开始所有的一切！站点没有维持您的 ViewState。

在 ASP .NET 中，当一个表单被提交时，表单会连同表单值一起出现在浏览器窗口中。如何做到的呢？这是因为 ASP .NET 维持了您的 ViewState。 ViewState 会在页面被提交到服务器时表明它的状态。这个状态是通过在带有 <form runat="server"> 控件的每个页面上放置一个隐藏域定义的。源代码如下所示：

```
<form name="_ctl0" method="post" action="page.aspx" id="_ctl0">
<input type="hidden" name="__VIEWSTATE"
value="dDwtNTI0ODU5MDE1Ozs+ZBCF2ryjMpeVgUrY2eTj79HNl4Q=" />

.....some code

</form>
```

维持 ViewState 是 ASP.NET Web Forms 的默认设置。如果您想不维持 ViewState，请在 .aspx 页面顶部包含指令 <%@ Page EnableViewState="false" %> ，或者向任意控件添加属性 EnableViewState="false" 。

请看下面的 .aspx 文件。它演示了"老"的运行方式。当您点击提交按钮，表单值将会消失：

## 实例

```
<html>
<body>

<form action="demo_classicasp.aspx" method="post">
Your name: <input type="text" name="fname" size="20">
<input type="submit" value="Submit">
</form>
<%
dim fname
fname=Request.Form("fname")
If fname<>"" Then
Response.Write("Hello " & fname & "!")
End If
%>

</body>
</html>
```

演示实例？

下面是新的 ASP .NET 方式。当您点击提交按钮，表单值不会消失：

# 实例

点击实例的右边框架中的查看源代码，您将看到 ASP .NET 已经在表单中添加了一个隐藏域来维持 ViewState。

```
<script runat="server">
Sub submit(sender As Object, e As EventArgs)
lbl1.Text="Hello " & txt1.Text & "!"
End Sub
</script>

<html>
<body>

<form runat="server">
Your name: <asp:TextBox id="txt1" runat="server" />
<asp:Button OnClick="submit" Text="Submit" runat="server" />
<p><asp:Label id="lbl1" runat="server" /></p>
</form>

</body>
</html>
```

演示实例？

# ASP.NET Web Forms - TextBox 控件

TextBox 控件用于创建用户可输入文本的文本框。

## TextBox 控件

TextBox 控件用于创建用户可输入文本的文本框。

TextBox 控件的特性和属性列在我们的 WebForms 控件参考手册页面。

下面的实例演示了您可能会用到的 TextBox 控件的一些属性：

## 实例

```
<html>
<body>

<form runat="server">

A basic TextBox:
<asp:TextBox id="tb1" runat="server" />
<br /><br />

A password TextBox:
<asp:TextBox id="tb2" TextMode="password" runat="server" />
<br /><br />

A TextBox with text:
<asp:TextBox id="tb4" Text="Hello World!" runat="server" />
<br /><br />

A multiline TextBox:
<asp:TextBox id="tb3" TextMode="multiline" runat="server" />
<br /><br />

A TextBox with height:
<asp:TextBox id="tb6" rows="5" TextMode="multiline"
runat="server" />
<br /><br />

A TextBox with width:
<asp:TextBox id="tb5" columns="30" runat="server" />

</form>

</body>
</html>
```

演示实例？

## 添加脚本

当表单被提交时，TextBox 控件的内容和设置可能会被服务器脚本修改。表单可通过点击一个按钮或当用户修改 TextBox 控件的值的时候进行提交。

在下面的实例中，我们在 .aspx 文件中声明了一个 TextBox 控件、一个 Button 控件和一个 Label 控件。当提交按钮被触发时，submit 子例程将被执行。submit 子例程将写入一行文本到 Label 控件中：

# 实例

```
<script runat="server">
Sub submit(sender As Object, e As EventArgs)
lbl1.Text="Your name is " & txt1.Text
End Sub
</script>

<html>
<body>

<form runat="server">
Enter your name:
<asp:TextBox id="txt1" runat="server" />
<asp:Button OnClick="submit" Text="Submit" runat="server" />
<p><asp:Label id="lbl1" runat="server" /></p>
</form>

</body>
</html>
```

演示实例？

在下面的实例中，我们在 .aspx 文件中声明了一个 TextBox 控件和一个 Label 控件。当您修改了 TextBox 中的值，并且在 TextBox 外部点击（或者按下了 Tab 键）时，change 子例程将会被执行。change 子例程将写入一行文本到 Label 控件中：

# 实例

```
<script runat="server">
Sub change(sender As Object, e As EventArgs)
lbl1.Text="You changed text to " & txt1.Text
End Sub
</script>

<html>
<body>

<form runat="server">
Enter your name:
<asp:TextBox id="txt1" runat="server"
text="Hello World!"
ontextchanged="change" autopostback="true"/>
<p><asp:Label id="lbl1" runat="server" /></p>
</form>

</body>
</html>
```

演示实例？

# ASP.NET Web Forms - Button 控件

Button 控件用于显示一个下压按钮。

## Button 控件

Button 控件用于显示一个下压按钮。下压按钮可能是一个提交按钮或者是一个命令按钮。在默认情况下，这个控件是提交按钮。

提交按钮没有命令名称，当它被点击时，它会把页面传回到服务器。您可以编写一些事件句柄，当提交按钮被点击时，用来控制动作的执行。

命令按钮有命令名称，并且允许您在页面上创建多个 Button 控件。您可以编写一些时间句柄，当命令按钮被点击时，用来控制动作的执行。

Button 控件的特性和属性列在我们的 WebForms 控件参考手册页面。

下面的实例演示了一个简单的 Button 控件：

```
<html>
<body>

<form runat="server">
<asp:Button id="b1" Text="Submit" runat="server" />
</form>

</body>
</html>
```

## 添加脚本

表单通常通过点击按钮进行提交。

在下面的实例中，我们在 .aspx 文件中声明了一个 TextBox 控件、一个 Button 控件和一个 Label 控件。当提交按钮被触发时，submit 子例程将被执行。submit 子例程将写入一行文本到 Label 控件中：

## 实例

```
<script runat="server">
Sub submit(sender As Object, e As EventArgs)
lbl1.Text="Your name is " & txt1.Text
End Sub
</script>

<html>
<body>

<form runat="server">
Enter your name:
<asp:TextBox id="txt1" runat="server" />
<asp:Button OnClick="submit" Text="Submit" runat="server" />
<p><asp:Label id="lbl1" runat="server" /></p>
</form>

</body>
</html>
```

演示实例？

# ASP.NET Web Forms - 数据绑定

我们可以使用数据绑定（Data Binding）来完成带可选项的列表，这些可选项来自某个导入的数据源，比如数据库、XML 文件或者脚本。

## 数据绑定

下面的控件是支持数据绑定的列表控件：

- asp:RadioButtonList
- asp:CheckBoxList
- asp:DropDownList
- asp:Listbox

以上每个控件的可选项通常是在一个或者多个 asp:ListItem 控件中定义，如下：

```
<html>
<body>

<form runat="server">
<asp:RadioButtonList id="countrylist" runat="server">
<asp:ListItem value="N" text="Norway" />
<asp:ListItem value="S" text="Sweden" />
<asp:ListItem value="F" text="France" />
<asp:ListItem value="I" text="Italy" />
</asp:RadioButtonList>
</form>

</body>
</html>
```

然而，我们可以使用某种独立的数据源进行数据绑定，比如数据库、XML 文件或者脚本，通过数据绑定来填充列表的可选项。

通过使用导入的数据源，数据从 HTML 中分离出来，并且对可选项的修改都是在独立的数据源中完成的。

在下面的三个章节中，我们将描述如何从脚本化的数据源中绑定数据。

# ASP.NET Web Forms - ArrayList 对象

ArrayList 对象是包含单个数据值的项目的集合。



## 尝试一下 - 实例

ArrayList DropDownList

ArrayList RadioButtonList

## 创建 ArrayList

ArrayList 对象是包含单个数据值的项目的集合。

通过 Add() 方法向 ArrayList 添加项目。

下面的代码创建了一个名为 mycountries 的 ArrayList 对象，并添加了四个项目：

```
<script runat="server">
Sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New ArrayList
mycountries.Add("Norway")
mycountries.Add("Sweden")
mycountries.Add("France")
mycountries.Add("Italy")
end if
end sub
</script>
```

在默认情况下，一个 ArrayList 对象包含 16 个条目。可通过 TrimToSize() 方法把 ArrayList 调整为最终尺寸：

```
<script runat="server">
Sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New ArrayList
mycountries.Add("Norway")
mycountries.Add("Sweden")
mycountries.Add("France")
mycountries.Add("Italy")
mycountries.TrimToSize()
end if
end sub
</script>
```

通过 Sort() 方法，ArrayList 也能够按照字母顺序或者数字顺序进行排序：

```
<script runat="server">
Sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New ArrayList
mycountries.Add("Norway")
mycountries.Add("Sweden")
mycountries.Add("France")
mycountries.Add("Italy")
mycountries.TrimToSize()
mycountries.Sort()
end if
end sub
</script>
```

要实现反向排序，请在 Sort() 方法后应用 Reverse() 方法：

```
<script runat="server">
Sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New ArrayList
mycountries.Add("Norway")
mycountries.Add("Sweden")
mycountries.Add("France")
mycountries.Add("Italy")
mycountries.TrimToSize()
mycountries.Sort()
mycountries.Reverse()
end if
end sub
</script>
```

# 绑定数据到 ArrayList

ArrayList 对象可为下列的控件自动生成文本和值：

- asp:RadioButtonList
- asp:CheckBoxList
- asp:DropDownList
- asp:Listbox

为了绑定数据到 RadioButtonList 控件，首先要在 .aspx 页面中创建一个 RadioButtonList 控件（不带任何 asp:ListItem 元素）：

```
<html>
<body>

<form runat="server">
<asp:RadioButtonList id="rb" runat="server" />
</form>

</body>
</html>
```

然后添加创建列表的脚本，并且绑定列表中的值到 RadioButtonList 控件：

# 实例

```
<script runat="server">
Sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New ArrayList
mycountries.Add("Norway")
mycountries.Add("Sweden")
mycountries.Add("France")
mycountries.Add("Italy")
mycountries.TrimToSize()
mycountries.Sort()
rb.DataSource=mycountries
rb.DataBind()
end if
end sub
</script>

<html>
<body>

<form runat="server">
<asp:RadioButtonList id="rb" runat="server" />
</form>

</body>
</html>
```

演示实例？

RadioButtonList 控件的 DataSource 属性被设置为该 ArrayList，它定义了这个 RadioButtonList 控件的数据源。RadioButtonList 控件的 DataBind() 方法把 RadioButtonList 控件与数据源绑定在一起。

注释：数据值作为控件的 Text 和 Value 属性来使用。如需添加不同于 Text 的 Value，请使用 Hashtable 对象或者 SortedList 对象。

# ASP.NET Web Forms - Hashtable 对象

Hashtable 对象包含用键/值对表示的项目。



## 尝试一下 - 实例

Hashtable RadiobuttonList 1

Hashtable RadiobuttonList 2

Hashtable DropDownList

## 创建 Hashtable

Hashtable 对象包含用键/值对表示的项目。键被用作索引，通过搜索键，可以实现对值的快速搜索。

通过 Add() 方法向 Hashtable 添加项目。

下面的代码创建了一个名为 mycountries 的 Hashtable 对象，并添加了四个元素：

```
<script runat="server">
Sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New Hashtable
mycountries.Add("N","Norway")
mycountries.Add("S","Sweden")
mycountries.Add("F","France")
mycountries.Add("I","Italy")
end if
end sub
</script>
```

## 数据绑定

Hashtable 对象可为下列的控件自动生成文本和值：

- asp:RadioButtonList
- asp:CheckBoxList
- asp:DropDownList
- asp:Listbox

为了绑定数据到 RadioButtonList 控件，首先要在 .aspx 页面中创建一个 RadioButtonList 控件（不带任何 asp:ListItem 元素）：

```
<html>
<body>

<form runat="server">
<asp:RadioButtonList id="rb" runat="server" AutoPostBack="True" />
</form>

</body>
</html>
```

然后添加创建列表的脚本，并且绑定列表中的值到 RadioButtonList 控件：

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New Hashtable
mycountries.Add("N","Norway")
mycountries.Add("S","Sweden")
mycountries.Add("F","France")
mycountries.Add("I","Italy")
rb.DataSource=mycountries
rb.DataValueField="Key"
rb.DataTextField="Value"
rb.DataBind()
end if
end sub
</script>

<html>
<body>

<form runat="server">
<asp:RadioButtonList id="rb" runat="server" AutoPostBack="True" />
</form>

</body>
</html>
```

然后我们添加一个子例程，当用户点击 RadioButtonList 控件中的某个项目时，该子例程会被执行。当某个单选按钮被点击时，label 中会出现一行文本：

# 实例

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New Hashtable
mycountries.Add("N","Norway")
mycountries.Add("S","Sweden")
mycountries.Add("F","France")
mycountries.Add("I","Italy")
rb.DataSource=mycountries
rb.DataValueField="Key"
rb.DataTextField="Value"
rb.DataBind()
end if
end sub

sub displayMessage(s as Object,e As EventArgs)
lbl1.text="Your favorite country is: " & rb.SelectedItem.Text
end sub
</script>

<html>
<body>

<form runat="server">
<asp:RadioButtonList id="rb" runat="server"
AutoPostBack="True" onSelectedIndexChanged="displayMessage" />
<p><asp:label id="lbl1" runat="server" /></p>
</form>

</body>
</html>
```

演示实例？

注释：您无法选择添加到 Hashtable 的项目的排序方式。如需对项目进行字母排序或者数字排序，请使用 SortedList 对象。

# ASP.NET Web Forms - SortedList 对象

SortedList 对象结合了 ArrayList 对象和 Hashtable 对象的特性。



## 尝试一下 - 实例

SortedList RadiobuttonList 1

SortedList RadiobuttonList 2

SortedList DropDownList

## SortedList 对象

SortedList 对象包含用键/值对表示的项目。SortedList 对象按照字母顺序或者数字顺序自动地对项目进行排序。

通过 Add() 方法向 SortedList 添加项目。通过 TrimToSize() 方法把 SortedList 调整为最终尺寸。

下面的代码创建了一个名为 mycountries 的 SortedList 对象，并添加了四个元素：

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New SortedList
mycountries.Add("N","Norway")
mycountries.Add("S","Sweden")
mycountries.Add("F","France")
mycountries.Add("I","Italy")
end if
end sub
</script>
```

## 数据绑定

SortedList 对象可为下列的控件自动生成文本和值：

- asp:RadioButtonList
- asp:CheckBoxList
- asp:DropDownList

- asp:Listbox

为了绑定数据到 RadioButtonList 控件，首先要在 .aspx 页面中创建一个 RadioButtonList 控件（不带任何 asp:ListItem 元素）：

```
<html>
<body>

<form runat="server">
<asp:RadioButtonList id="rb" runat="server" AutoPostBack="True" />
</form>

</body>
</html>
```

然后添加创建列表的脚本，并且绑定列表中的值到 RadioButtonList 控件：

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New SortedList
mycountries.Add("N","Norway")
mycountries.Add("S","Sweden")
mycountries.Add("F","France")
mycountries.Add("I","Italy")
rb.DataSource=mycountries
rb.DataValueField="Key"
rb.DataTextField="Value"
rb.DataBind()
end if
end sub
</script>

<html>
<body>

<form runat="server">
<asp:RadioButtonList id="rb" runat="server" AutoPostBack="True" />
</form>

</body>
</html>
```

然后我们添加一个子例程，当用户点击 RadioButtonList 控件中的某个项目时，该子例程会被执行。当某个单选按钮被点击时，label 中会出现一行文本：

# 实例

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New SortedList
mycountries.Add("N","Norway")
mycountries.Add("S","Sweden")
mycountries.Add("F","France")
mycountries.Add("I","Italy")
rb.DataSource=mycountries
rb.DataValueField="Key"
rb.DataTextField="Value"
rb.DataBind()
end if
end sub

sub displayMessage(s as Object,e As EventArgs)
lbl1.text="Your favorite country is: " & rb.SelectedItem.Text
end sub
</script>

<html>
<body>

<form runat="server">
<asp:RadioButtonList id="rb" runat="server"
AutoPostBack="True" onSelectedIndexChanged="displayMessage" />
<p><asp:label id="lbl1" runat="server" /></p>
</form>

</body>
</html>
```

演示实例？

# ASP.NET Web Forms - XML 文件

我们可以绑定 XML 文件到列表控件。

## 一个 XML 文件

这里有一个名为 "countries.xml" 的 XML 文件：

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<countries>

<country>
<text>Norway</text>
<value>N</value>
</country>

<country>
<text>Sweden</text>
<value>S</value>
</country>

<country>
<text>France</text>
<value>F</value>
</country>

<country>
<text>Italy</text>
<value>I</value>
</country>

</countries>
```

查看这个 XML 文件：countries.xml

## 绑定 DataSet 到 List 控件

首先，导入 "System.Data" 命名空间。我们需要该命名空间与 DataSet 对象一起工作。把下面这条指令包含在 .aspx 页面的顶部：

```
<%@ Import Namespace="System.Data" %>
```

接着，为 XML 文件创建一个 DataSet，并在页面第一次加载时把这个 XML 文件载入 DataSet：

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New DataSet
mycountries.ReadXml(MapPath("countries.xml"))
end if
end sub
```

为了绑定数据到 RadioButtonList 控件，首先要在 .aspx 页面中创建一个 RadioButtonList 控件（不带任何 asp:ListItem 元素）：

```
<html>
<body>

<form runat="server">
<asp:RadioButtonList id="rb" runat="server" AutoPostBack="True" />
</form>

</body>
</html>
```

然后添加创建 XML DataSet 的脚本，并且绑定 XML DataSet 中的值到 RadioButtonList 控件：

```
<%@ Import Namespace="System.Data" %>

<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New DataSet
mycountries.ReadXml(MapPath("countries.xml"))
rb.DataSource=mycountries
rb.DataValueField="value"
rb.DataTextField="text"
rb.DataBind()
end if
end sub
</script>

<html>
<body>

<form runat="server">
<asp:RadioButtonList id="rb" runat="server"
AutoPostBack="True" onSelectedIndexChanged="displayMessage" />
</form>

</body>
</html>
```

然后我们添加一个子例程，当用户点击 RadioButtonList 控件中的某个项目时，该子例程会被执行。当某个单选按钮被点击时，label 中会出现一行文本：

# 实例

```
<%@ Import Namespace="System.Data" %>

<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycountries=New DataSet
mycountries.ReadXml(MapPath("countries.xml"))
rb.DataSource=mycountries
rb.DataValueField="value"
rb.DataTextField="text"
rb.DataBind()
end if
end sub

sub displayMessage(s as Object,e As EventArgs)
lbl1.text="Your favorite country is: " & rb.SelectedItem.Text
end sub
</script>

<html>
<body>

<form runat="server">
<asp:RadioButtonList id="rb" runat="server"
AutoPostBack="True" onSelectedIndexChanged="displayMessage" />
<p><asp:label id="lbl1" runat="server" /></p>
</form>

</body>
</html>
```

演示实例？

# ASP.NET Web Forms - Repeater 控件

Repeater 控件用于显示被绑定在该控件上的项目的重复列表。

## 绑定 DataSet 到 Repeater 控件

Repeater 控件用于显示被绑定在该控件上的项目的重复列表。Repeater 控件可被绑定到数据库表、XML 文件或者其他项目列表。在这里，我们将演示如何绑定 XML 文件到 Repeater 控件。

在我们的实例中，我们将使用下面的 XML 文件（"cdcatalog.xml"）：

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<catalog>
<cd>
<title>Empire Burlesque</title>
<artist>Bob Dylan</artist>
<country>USA</country>
<company>Columbia</company>
<price>10.90</price>
<year>1985</year>
</cd>
<cd>
<title>Hide your heart</title>
<artist>Bonnie Tyler</artist>
<country>UK</country>
<company>CBS Records</company>
<price>9.90</price>
<year>1988</year>
</cd>
<cd>
<title>Greatest Hits</title>
<artist>Dolly Parton</artist>
<country>USA</country>
<company>RCA</company>
<price>9.90</price>
<year>1982</year>
</cd>
<cd>
<title>Still got the blues</title>
<artist>Gary Moore</artist>
<country>UK</country>
<company>Virgin records</company>
<price>10.20</price>
<year>1990</year>
</cd>
<cd>
<title>Eros</title>
<artist>Eros Ramazzotti</artist>
<country>EU</country>
<company>BMG</company>
<price>9.90</price>
<year>1997</year>
</cd>
</catalog>
```

查看这个 XML 文件：cdcatalog.xml

首先，导入 "System.Data" 命名空间。我们需要该命名空间与 DataSet 对象一起工作。 把下面这条指令包含在 .aspx 页面的顶部：

```
<%@ Import Namespace="System.Data" %>
```

接着，为 XML 文件创建一个 DataSet，并在页面第一次加载时把这个 XML 文件载入 DataSet：

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycdcatalog=New DataSet
mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
end if
end sub
```

然后我们在 .aspx 页面中创建一个 Repeater 控件。<HeaderTemplate> 元素中的内容被首先呈现，并且在输出中仅出现一次，而 <ItemTemplate> 元素中的内容会对应 DataSet 中的每条 "record" 重复出现，最后，<FooterTemplate> 元素中的内容在输出中仅出现一次：

```
<html>
<body>

<form runat="server">
<asp:Repeater id="cdcatalog" runat="server">

<HeaderTemplate>
...
</HeaderTemplate>

<ItemTemplate>
...
</ItemTemplate>

<FooterTemplate>
...
</FooterTemplate>

</asp:Repeater>
</form>

</body>
</html>
```

然后我们添加创建 DataSet 的脚本，并且绑定 mycdcatalog DataSet 到 Repeater 控件。然后使用 HTML 标签来填充 Repeater 控件，并通过 <%#Container.DataItem("fieldname")%> 绑定数据项目到 <ItemTemplate> 区域内的单元格中：

# 实例

```
<%@ Import Namespace="System.Data" %>

<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycdcatalog=New DataSet
mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
cdcatalog.DataSource=mycdcatalog
cdcatalog.DataBind()
end if
end sub
</script>

<html>
<body>

<form runat="server">
<asp:Repeater id="cdcatalog" runat="server">

<HeaderTemplate>
<table border="1" width="100%">
<tr>
<th>Title</th>
<th>Artist</th>
<th>Country</th>
<th>Company</th>
<th>Price</th>
<th>Year</th>
</tr>
</HeaderTemplate>

<ItemTemplate>
<tr>
<td><%#Container.DataItem("title")%></td>
<td><%#Container.DataItem("artist")%></td>
<td><%#Container.DataItem("country")%></td>
<td><%#Container.DataItem("company")%></td>
<td><%#Container.DataItem("price")%></td>
<td><%#Container.DataItem("year")%></td>
</tr>
</ItemTemplate>

<FooterTemplate>
</table>
</FooterTemplate>

</asp:Repeater>
</form>

</body>
</html>
```

演示实例？

# 使用 <AlternatingItemTemplate>

您可以在 <ItemTemplate> 元素后添加 <AlternatingItemTemplate> 元素，用来描述输出中交替行的外观。在下面的实例中，表格每隔一行就会显示为浅灰色的背景：

## 实例

```
<%@ Import Namespace="System.Data" %>

<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycdcatalog=New DataSet
mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
cdcatalog.DataSource=mycdcatalog
cdcatalog.DataBind()
end if
end sub
</script>

<html>
<body>

<form runat="server">
<asp:Repeater id="cdcatalog" runat="server">

<HeaderTemplate>
<table border="1" width="100%">
<tr>
<th>Title</th>
<th>Artist</th>
<th>Country</th>
<th>Company</th>
<th>Price</th>
<th>Year</th>
</tr>
</HeaderTemplate>

<ItemTemplate>
<tr>
<td><%#Container.DataItem("title")%></td>
<td><%#Container.DataItem("artist")%></td>
<td><%#Container.DataItem("country")%></td>
<td><%#Container.DataItem("company")%></td>
<td><%#Container.DataItem("price")%></td>
<td><%#Container.DataItem("year")%></td>
</tr>
</ItemTemplate>

<AlternatingItemTemplate>
<tr bgcolor="#e8e8e8">
<td><%#Container.DataItem("title")%></td>
<td><%#Container.DataItem("artist")%></td>
<td><%#Container.DataItem("country")%></td>
<td><%#Container.DataItem("company")%></td>
<td><%#Container.DataItem("price")%></td>
<td><%#Container.DataItem("year")%></td>
</tr>
</AlternatingItemTemplate>

<FooterTemplate>
</table>
</FooterTemplate>

</asp:Repeater>
</form>

</body>
</html>
```

演示实例？

# 使用 <SeparatorTemplate>

<SeparatorTemplate> 元素用于描述每个记录之间的分隔符。在下面的实例中，每个表格行之间插入了一条水平线：

# 实例

```
<%@ Import Namespace="System.Data" %>

<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycdcatalog=New DataSet
mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
cdcatalog.DataSource=mycdcatalog
cdcatalog.DataBind()
end if
end sub
</script>

<html>
<body>

<form runat="server">
<asp:Repeater id="cdcatalog" runat="server">

<HeaderTemplate>
<table border="0" width="100%">
<tr>
<th>Title</th>
<th>Artist</th>
<th>Country</th>
<th>Company</th>
<th>Price</th>
<th>Year</th>
</tr>
</HeaderTemplate>

<ItemTemplate>
<tr>
<td><%#Container.DataItem("title")%></td>
<td><%#Container.DataItem("artist")%></td>
<td><%#Container.DataItem("country")%></td>
<td><%#Container.DataItem("company")%></td>
<td><%#Container.DataItem("price")%></td>
<td><%#Container.DataItem("year")%></td>
</tr>
</ItemTemplate>

<SeparatorTemplate>
<tr>
<td colspan="6"><hr /></td>
</tr>
</SeparatorTemplate>

<FooterTemplate>
</table>
</FooterTemplate>

</asp:Repeater>
</form>

</body>
</html>
```

演示实例？

# ASP.NET Web Forms - DataList 控件

DataList 控件，类似于 Repeater 控件，用于显示绑定在该控件上的项目的重复列表。不过，DataList 控件会默认地在数据项目上添加表格。

## 绑定 DataSet 到 DataList 控件

DataList 控件，类似于 Repeater 控件，用于显示绑定在该控件上的项目的重复列表。不过，DataList 控件会默认地在数据项目上添加表格。DataList 控件可被绑定到数据库表、XML 文件或者其他项目列表。在这里，我们将演示如何绑定 XML 文件到 DataList 控件。

在我们的实例中，我们将使用下面的 XML 文件（"cdcatalog.xml"）：

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<catalog>
<cd>
<title>Empire Burlesque</title>
<artist>Bob Dylan</artist>
<country>USA</country>
<company>Columbia</company>
<price>10.90</price>
<year>1985</year>
</cd>
<cd>
<title>Hide your heart</title>
<artist>Bonnie Tyler</artist>
<country>UK</country>
<company>CBS Records</company>
<price>9.90</price>
<year>1988</year>
</cd>
<cd>
<title>Greatest Hits</title>
<artist>Dolly Parton</artist>
<country>USA</country>
<company>RCA</company>
<price>9.90</price>
<year>1982</year>
</cd>
<cd>
<title>Still got the blues</title>
<artist>Gary Moore</artist>
<country>UK</country>
<company>Virgin records</company>
<price>10.20</price>
<year>1990</year>
</cd>
<cd>
<title>Eros</title>
<artist>Eros Ramazzotti</artist>
<country>EU</country>
<company>BMG</company>
<price>9.90</price>
<year>1997</year>
</cd>
</catalog>
```

查看这个 XML 文件：cdcatalog.xml

首先，导入 "System.Data" 命名空间。我们需要该命名空间与 DataSet 对象一起工作。 把下面这条指令包含在 .aspx 页面的顶部：

```
<%@ Import Namespace="System.Data" %>
```

接着，为 XML 文件创建一个 DataSet，并在页面第一次加载时把这个 XML 文件载入 DataSet：

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycdcatalog=New DataSet
mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
end if
end sub
```

然后我们在 .aspx 页面中创建一个 DataList 控件。<HeaderTemplate> 元素中的内容被首先呈现，并且在输出中仅出现一次，而 <ItemTemplate> 元素中的内容会对应 DataSet 中的每条 "record" 重复出现，最后，<FooterTemplate> 元素中的内容在输出中仅出现一次：

```
<html>
<body>

<form runat="server">
<asp:DataList id="cdcatalog" runat="server">

<HeaderTemplate>
...
</HeaderTemplate>

<ItemTemplate>
...
</ItemTemplate>

<FooterTemplate>
...
</FooterTemplate>

</asp:DataList>
</form>

</body>
</html>
```

然后我们添加创建 DataSet 的脚本，并且绑定 mycdcatalog DataSet 到 DataList 控件。然后使用包含表头的 <HeaderTemplate>、包含要显示的数据项的 <ItemTemplate> 和包含文本的 <FooterTemplate> 来填充 DataList 控件。请注意，可设置 DataList 的 gridlines 属性为 "both" 来显示表格边框：

## 实例

```
<%@ Import Namespace="System.Data" %>

<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycdcatalog=New DataSet
mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
cdcatalog.DataSource=mycdcatalog
cdcatalog.DataBind()
end if
end sub
</script>

<html>
<body>

<form runat="server">
<asp:DataList id="cdcatalog"
gridlines="both" runat="server">

<HeaderTemplate>
My CD Catalog
</HeaderTemplate>

<ItemTemplate>
"<%#Container.DataItem("title")%>" of
<%#Container.DataItem("artist")%> -
$<%#Container.DataItem("price")%>
</ItemTemplate>

<FooterTemplate>
Copyright Hege Refsnes
</FooterTemplate>

</asp:DataList>
</form>

</body>
</html>
```

演示实例？

# 使用样式

您也可以向 DataList 控件添加样式，让输出更加花哨：

# 实例

```
<%@ Import Namespace="System.Data" %>

<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycdcatalog=New DataSet
mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
cdcatalog.DataSource=mycdcatalog
cdcatalog.DataBind()
end if
end sub
</script>

<html>
<body>

<form runat="server">
<asp:DataList id="cdcatalog"
runat="server"
cellpadding="2"
cellspacing="2"
borderstyle="inset"
backcolor="#e8e8e8"
width="100%"
headerstyle-font-name="Verdana"
headerstyle-font-size="12pt"
headerstyle-horizontalalign="center"
headerstyle-font-bold="true"
itemstyle-backcolor="#778899"
itemstyle-forecolor="#ffffff"
footerstyle-font-size="9pt"
footerstyle-font-italic="true">

<HeaderTemplate>
My CD Catalog
</HeaderTemplate>

<ItemTemplate>
"<%#Container.DataItem("title")%>" of
<%#Container.DataItem("artist")%> -
$<%#Container.DataItem("price")%>
</ItemTemplate>

<FooterTemplate>
Copyright Hege Refsnes
</FooterTemplate>

</asp:DataList>
</form>

</body>
</html>
```

演示实例？

# 使用 <AlternatingItemTemplate>

您可以在 <ItemTemplate> 元素后添加 <AlternatingItemTemplate> 元素，用来描述输出中交替行的外观。您可以在 DataList 控件内部对 <AlternatingItemTemplate> 区域的数据添加样式：

# 实例

```
<%@ Import Namespace="System.Data" %>

<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
dim mycdcatalog=New DataSet
mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
cdcatalog.DataSource=mycdcatalog
cdcatalog.DataBind()
end if
end sub
</script>

<html>
<body>

<form runat="server">
<asp:DataList id="cdcatalog"
runat="server"
cellpadding="2"
cellspacing="2"
borderstyle="inset"
backcolor="#e8e8e8"
width="100%"
headerstyle-font-name="Verdana"
headerstyle-font-size="12pt"
headerstyle-horizontalalign="center"
headerstyle-font-bold="True"
itemstyle-backcolor="#778899"
itemstyle-forecolor="#ffffff"
alternatingitemstyle-backcolor="#e8e8e8"
alternatingitemstyle-forecolor="#000000"
footerstyle-font-size="9pt"
footerstyle-font-italic="True">

<HeaderTemplate>
My CD Catalog
</HeaderTemplate>

<ItemTemplate>
"<%#Container.DataItem("title")%>" of
<%#Container.DataItem("artist")%> -
$<%#Container.DataItem("price")%>
</ItemTemplate>

<AlternatingItemTemplate>
"<%#Container.DataItem("title")%>" of
<%#Container.DataItem("artist")%> -
$<%#Container.DataItem("price")%>
</AlternatingItemTemplate>

<FooterTemplate>
&copy; Hege Refsnes
</FooterTemplate>

</asp:DataList>
</form>

</body>
</html>
```

演示实例？

# ASP.NET Web Forms - 数据库连接

ADO.NET 也是 .NET 框架的组成部分。ADO.NET 用于处理数据访问。通过 ADO.NET，您可以操作数据库。



## 尝试一下 - 实例

数据库连接 - 绑定到 DataList 控件

数据库连接 - 绑定到 Repeater 控件

## 什么是 ADO.NET？

- ADO.NET 是 .NET 框架的组成部分
- ADO.NET 由一系列用于处理数据访问的类组成
- ADO.NET 完全基于 XML
- ADO.NET 没有 Recordset 对象，这一点与 ADO 不同

## 创建数据库连接

在我们的实例中，我们将使用 Northwind 数据库。

首先，导入 "System.Data.OleDb" 命名空间。我们需要这个命名空间来操作 Microsoft Access 和其他 OLE DB 数据库提供商。我们将在 Page_Load 子例程中创建这个数据库的连接。我们创建一个 dbconn 变量，并为其赋值一个新的 OleDbConnection 类，这个类带有指示 OLE DB 提供商和数据库位置的连接字符串。然后我们打开数据库连接：

```
<%@ Import Namespace="System.Data.OleDb" %>

<script runat="server">
sub Page_Load
dim dbconn
dbconn=New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;
data source=" & server.mappath("northwind.mdb"))
dbconn.Open()
end sub
</script>
```

注释：这个连接字符串必须是没有折行的连续字符串！

# 创建数据库命令

为了指定需从数据库取回的记录，我们将创建一个 dbcomm 变量，并为其赋值一个新的 OleDbCommand 类。这个 OleDbCommand 类用于发出针对数据库表的 SQL 查询：

```
<%@ Import Namespace="System.Data.OleDb" %>

<script runat="server">
sub Page_Load
dim dbconn,sql,dbcomm
dbconn=New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;
data source=" & server.mappath("northwind.mdb"))
dbconn.Open()
sql="SELECT * FROM customers"
dbcomm=New OleDbCommand(sql,dbconn)
end sub
</script>
```

# 创建 DataReader

OleDbDataReader 类用于从数据源中读取记录流。DataReader 是通过调用 OleDbCommand 对象的 ExecuteReader 方法来创建的：

```
<%@ Import Namespace="System.Data.OleDb" %>

<script runat="server">
sub Page_Load
dim dbconn,sql,dbcomm,dbread
dbconn=New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;
data source=" & server.mappath("northwind.mdb"))
dbconn.Open()
sql="SELECT * FROM customers"
dbcomm=New OleDbCommand(sql,dbconn)
dbread=dbcomm.ExecuteReader()
end sub
</script>
```

# 绑定到 Repeater 控件

然后，我们绑定 DataReader 到 Repeater 控件：

# 实例

```
<%@ Import Namespace="System.Data.OleDb" %>

<script runat="server">
sub Page_Load
dim dbconn,sql,dbcomm,dbread
dbconn=New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;
data source=" & server.mappath("northwind.mdb"))
dbconn.Open()
sql="SELECT * FROM customers"
dbcomm=New OleDbCommand(sql,dbconn)
dbread=dbcomm.ExecuteReader()
customers.DataSource=dbread
customers.DataBind()
dbread.Close()
dbconn.Close()
end sub
</script>

<html>
<body>

<form runat="server">
<asp:Repeater id="customers" runat="server">

<HeaderTemplate>
<table border="1" width="100%">
<tr>
<th>Companyname</th>
<th>Contactname</th>
<th>Address</th>
<th>City</th>
</tr>
</HeaderTemplate>

<ItemTemplate>
<tr>
<td><%#Container.DataItem("companyname")%></td>
<td><%#Container.DataItem("contactname")%></td>
<td><%#Container.DataItem("address")%></td>
<td><%#Container.DataItem("city")%></td>
</tr>
</ItemTemplate>

<FooterTemplate>
</table>
</FooterTemplate>

</asp:Repeater>
</form>

</body>
</html>
```

演示实例？

# 关闭数据库连接

如果不再需要访问数据库，请记得关闭 DataReader 和数据库连接：

```
dbread.Close()
dbconn.Close()
```

# ASP.NET Web Forms - 母版页

母版页为您的网站的其他页面提供模版。

## 母版页

母版页允许您为您的 web 应用程序中的所有页面（或页面组）创建一致的外观和行为。

母版页为其他页面提供模版，带有共享的布局和功能。母版页为内容定义了可被内容页覆盖的占位符。输出结果是母版页和内容页的组合。

内容页包含您想要显示的内容。

当用户请求内容页时，ASP.NET 会对页面进行合并以生成结合了母版页布局和内容页内容的输出。

## 母版页实例

```
<%@ Master %>

<html>
<body>
<h1>Standard Header From Masterpage</h1>
<asp:ContentPlaceHolder id="CPH1" runat="server">
</asp:ContentPlaceHolder>
</body>
</html>
```

上面的母版页是一个为其他页面设计的普通 HTML 模版页。

**@ Master** 指令定义它为一个母版页。

母版页为单独的内容包含占位标签 **<asp:ContentPlaceHolder>**。

**id="CPH1"** 属性标识占位符，在相同母版页中允许多个占位符。

这个母版页被保存为 **"master1.master"**。

 注释：母版页也能够包含代码，允许动态的内容。

## 内容页实例

```
<%@ Page MasterPageFile="master1.master" %>

<asp:Content ContentPlaceHolderId="CPH1" runat="server">
<h2>Individual Content</h2>
<p>Paragraph 1</p>
<p>Paragraph 2</p>
</asp:Content>
```

上面的内容页是站点中独立的内容页中的一个。

**@ Page** 指令定义它为一个标准的内容页。

内容页包含内容标签 **<asp:Content>**，该标签引用了母版页（ContentPlaceHolderId="CPH1"）。

这个内容页被保存为 **"mypage1.aspx"**。

当用户请求该页面时，ASP.NET 就会将母版页与内容页进行合并。

点击这里显示 mypage1.aspx

>  注释：内容文本必须位于 <asp:Content> 标签内部。标签外的内容文本是不允许的。

# 带控件的内容页

```
<%@ Page MasterPageFile="master1.master" %>

<asp:Content ContentPlaceHolderId="CPH1" runat="server">
<h2>W3CSchool</h2>
<form runat="server">
<asp:TextBox id="textbox1" runat="server" />
<asp:Button id="button1" runat="server" text="Button" />
</form>
</asp:Content>
```

上面的内容页演示了如何把 .NET 控件插入内容页，就像插入一个普通的页面中。

点击这里显示 mypage2.aspx

# ASP.NET Web Forms - 导航

ASP.NET 带有内建的导航控件。

## 网站导航

维护大型网站的菜单是困难而且费时的。

在 ASP.NET 中，菜单可存储在文件中，这样易于维护。文件通常名为 **web.sitemap**，并且被存放在网站的根目录下。

此外，ASP.NET 有三个心的导航控件：

- Dynamic menus
- TreeViews
- Site Map Path

## Sitemap 文件

在本教程中，使用下面的 sitemap 文件：

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<siteMap>
<siteMapNode title="Home" url="/aspnet/w3home.aspx">
<siteMapNode title="Services" url="/aspnet/w3services.aspx">
<siteMapNode title="Training" url="/aspnet/w3training.aspx"/>
<siteMapNode title="Support" url="/aspnet/w3support.aspx"/>
</siteMapNode>
</siteMapNode>
</siteMap>
```

创建 sitemap 文件的规则：

- XML 文件必须包含 围绕内容的 <siteMap> 标签
- <siteMap> 标签只能有一个 <siteMapNode> 子节点（"home" 页面）
- 每个 <siteMapNode> 可以有多个子节点（网页）
- 每个 <siteMapNode> 带有定义页面标题和 URL 的属性



注释：sitemap 文件必须位于站点根目录下，URL 属性必须相对于该根目录。

## 动态菜单

<asp:Menu> 控件可显示标准的站点导航菜单。

代码实例：

```
<asp:SiteMapDataSource id="nav1" runat="server" />

<form runat="server">
<asp:Menu runat="server" DataSourceId="nav1" />
</form>
```

上面实例中的 **<asp:Menu>** 控件是一个供服务器创建导航菜单的占位符。

控件的数据源由 **DataSourceId** 属性定义。 **id="nav1"** 把数据源连接到 **<asp:SiteMapDataSource>** 控件。

**<asp:SiteMapDataSource>** 控件自动连接默认的 sitemap 文件（**web.sitemap**）。

# TreeView

<asp:TreeView> 控件可显示多级导航菜单。

这种菜单看上去像一棵带有枝叶的树，可通过 + 或 - 符号来打开或关闭。

代码实例：

```
<asp:SiteMapDataSource id="nav1" runat="server" />

<form runat="server">
<asp:TreeView runat="server" DataSourceId="nav1" />
</form>
```

上面实例中的 **<asp:TreeView>** 控件是一个供服务器创建导航菜单的占位符。

控件的数据源由 **DataSourceId** 属性定义。 **id="nav1"** 把数据源连接到 **<asp:SiteMapDataSource>** 控件。

**<asp:SiteMapDataSource>** 控件自动连接默认的 sitemap 文件（**web.sitemap**）。

# SiteMapPath

SiteMapPath 控件可显示指向当前页面的指针（导航路径）。该路径显示为指向上级页面的可点击链接。

与 TreeView 和 Menu 控件不同，SiteMapPath 控件不使用 SiteMapDataSource。SiteMapPath 控件默认使用 web.sitemap 文件。

>  提示：如果 SiteMapPath 没有正确显示，很可能是由于 web.sitemap 文件中存在 URL 错误（打印错误）。

代码实例：

```
<form runat="server">
<asp:SiteMapPath runat="server" />
</form>
```

上面实例中的 **<asp:SiteMapPath>** 控件是一个供服务器创建导航菜单的占位符。

# Web Pages 参考手册

# ASP.NET Web Pages - 类

## ASP.NET 类参考手册

| 方法 | 描述 |
| --- | --- |
| AsBool(), AsBool(true\|false) | 转换字符串值为布尔值（true/false）。如果字不能转换为true/false，回 false 或者其他规定的 |
| AsDateTime(), AsDateTime(*value*) | 转换字符串值为日期/时返回 DateTime。如果字不能转换为日期/时间，回 MinValue 或者其他规值。 |
| AsDecimal(), AsDecimal(*value*) | 转换字符串值为十进制如果字符串不能转换为制值，则返回 0.0 或者规定的值。 |
| AsFloat(), AsFloat(*value*) | 转换字符串值为浮点数果字符串不能转换为浮数，则返回 0.0 或者其定的值。 |
| AsInt(), AsInt(*value*) | 转换字符串值为整数。字符串不能转换成整数，返回 0 或者其他规定的 |
| Href(*path [, param1 [, param2]]*) | 从带有可选的附加路径的本地文件路径创建一览器兼容的 URL。 |
| Html.Raw(*value*) | Renders *value* 呈现为标记，而不是呈现为 H编码输出。 |
| IsBool(), IsDateTime(), IsDecimal(), IsFloat(), IsInt() | 如果该值可以从字符串为指定的类型，则返回 true。 |
| IsEmpty() | 如果对象或者变量没有则返回 true。 |
| IsPost | 如果请求是 POST，则true。（初始请求通常GET。） |
| Layout | 规定布局页面的路径应此页面。 |

| | |
|---|---|
| PageData[*key*], PageData[*index*], Page | 在当前请求的页面、布局页面、部分页面之间包含数据。您可以使用动态来对相同的数据进行属性问。 |
| RenderBody() | (Layout pages) 呈现没布局页面任何命名区域容页的内容Renders the content of a content pa that is not in any name sections. |
| RenderPage(*path*, *values*) RenderPage(*path[, param1 [, param2]]*) | 呈现使用了规定的路径选的额外数据的内容页可以通过 position（实或者 key（实例 2）从 PageData 获取额外参数值。 |
| RenderSection(*sectionName [, required* = true\|false*]*) | (Layout pages) 呈现一名字的内容区域。设置 *required* 让一个区域为非可选的。 |
| Request.Cookies[*key*] | 获取或者设置 HTTP co 的值。 |
| Request.Files[*key*] | Gets 在当前请求中上传件。 |
| Request.Form[*key*] | 获取在表单中 post 的数（作为字符串）。Request.Form 和 Request.QueryString 者 [*key*] 检查。 |
| Request.QueryString[*key*] | 获取 URL 查询字符串中的数据。Request.Form Request.QueryString 者 [*key*] 检查。 |
| Request.Unvalidated(*key*) Request.Unvalidated().QueryString\|Form\|Cookies\|Headers[*key*] | 有选择地禁用请求验证单元素、查询字符串值、cookie、header 值）。验证默认是开启的，防户提交标记或者其他潜危险内容。 |
| Response.AddHeader(*name*, *value*) | 在应答中添加一个 HTT 务器响应头。 |
| Response.OutputCache(*seconds [, sliding] [, varyByParams]*) | Caches 在指定时间的出缓存。设置 *sliding* 来每个页面的访问超时时设置 *varyByParams* 为 |

| | |
|---|---|
| | 页面的每个不同的查询字<br>串缓存不同版本的页面。 |
| Response.Redirect(*path*) | 重定向浏览器请求到一个<br>的位置。 |
| Response.SetStatus(*httpStatusCode*) | 设置HTTP状态代码发送<br>览器。 |
| Response.WriteBinary(*data [, mimetype]*) | 写入 *data* 内容响应可选<br>MIME类型。 |
| Response.WriteFile(*file*) | 写入文件内容响应。 |
| @section(*sectionName*) { *content* } | （布局页面）定义一个名<br>字的内容区域。 |
| Server.HtmlDecode(*htmlText*) | 解码一个HTML编码的字<br>串。 |
| Server.HtmlEncode(*text*) | 为呈现在 HTML 标记中<br>符串编码。 |
| Server.MapPath(*virtualPath*) | 为指定的虚拟路径返回服<br>器的物理路径。 |
| Server.UrlDecode(*urlText*) | 解码URL文本。 |
| Server.UrlEncode(*text*) | URL文本编码。 |
| Session[*key*] | 获取或设置一个存在的值<br>直到用户关闭浏览器。 |
| ToString() | 显示一个用字符串表示的<br>象的值。 |
| UrlData[*index*] | 从 URL 获取额外的数据<br>如，*/MyPage/ExtraData* |

# ASP.NET Web Pages - WebSecurity 对象

## 描述

**WebSecurity** 对象提供 ASP.NET Web Pages 应用程序的安全性和认证。

通过 WebSecurity 对象，您可以创建用户帐户，登录和注销用户，重置或者更改密码，以及其他更多与安全性相关的功能。

## WebSecurity 对象参考手册 - 属性

| 属性 | 描述 |
| --- | --- |
| CurrentUserId | 获取当前登录用户的 ID。 |
| CurrentUserName | 获取当前登录用户的名称。 |
| HasUserId | 如果当前有用户 ID，则返回 true。 |
| IsAuthenticated | 如果当前用户是登录的，则返回 true。 |

## WebSecurity 对象参考手册 - 方法

| 方法 | 描述 |
| --- | --- |
| ChangePassword() | 为指定的用户更改密码。 |
| ConfirmAccount() | 使用帐户确认令牌确认帐户。 |
| CreateAccount() | 创建一个新的用户帐户。 |
| CreateUserAndAccount() | 创建一个新的用户帐户。 |
| GeneratePasswordResetToken() | 生成一个密码重置令牌，可以在电子邮件中发送给用户以便用户可以重设密码。 |
| GetCreateDate() | 获取指定会员创建的时间。 |
| GetPasswordChangeDate() | 获取密码变更的日期和时间。 |
| GetUserId() | 根据用户名称获取用户 ID。 |
| InitializeDatabaseConnection() | 初始化 WebSecurity 系统（数据库）。 |
| IsConfirmed() | 检查用户是否已被确认。如果已确认，则返回 true。（例如，可通过电子邮件进行确认。） |
| IsCurrentUser() | 检查当前用户的名称是否与指定用户名匹配。如果匹配，则返回 true。 |
| Login() | 设置身份验证令牌，登录用户。 |
| Logout() | 移除身份验证令牌，注销用户。 |
| RequireAuthenticatedUser() | 如果用户未通过身份验证，则设置 HTTP 状态为 401（未经授权）。 |
| RequireRoles() | 如果当前用户不是指定角色的成员，则设置 HTTP 状态为 401（未经授权）。 |
| RequireUser() | 如果当前用户不是指定用户名的用户，则设置 HTTP 状态为 401（未经授权）。 |
| ResetPassword() | 如果密码重置令牌是有效的，改变用户的密码为新密码。 |
| UserExists() | 检查指定的用户是否存在。 |

# 技术数据

| 名称 | 值 |
| --- | --- |
| Class | WebMatrix.WebData.WebSecurity |
| Namespace | WebMatrix.WebData |
| Assembly | WebMatrix.WebData.dll |

# 初始化 WebSecurity 数据库

如果您想在您的代码中使用 WebSecurity 对象，首先您必须创建或者初始化 WebSecurity 数据库。

在您的 Web 根目录下，创建一个名为 **_AppStart.cshtml** 的页面（如果已存在，则直接编辑页面）。

将下面的代码复制到文件中：

# _AppStart.cshtml

```
@{
WebSecurity.InitializeDatabaseConnection("Users", "UserProfile", "UserId", "Email", true)
}
```

上面的代码将在每次网站（应用程序）启动时运行。它初始化了 WebSecurity 数据库。

**"Users"** 是 WebSecurity 数据库（Users.sdf）的名称。

**"UserProfile"** 是包含用户配置信息的数据库表的名称。

**"UserId"** 是包含用户 ID（主键）的列的名称。

**"Email"** 是包含用户名的列的名称。

最后一个参数 **true** 是一个布尔值，表示如果用户配置表和会员表不存在，则会自动创建表。如果不想自动创建表，应设置参数为 **false**。

>  虽然 **true** 表示自动创建数据库 表，但是数据库不会被自动创建。所以数据库必须存在。 |

# WebSecurity 数据库

**UserProfile** 表为每个用户创建保存一条记录，用户 ID（主键）和用户名字（email）：

| UserId | Email |
|---|---|
| 1 | john@johnson.net |
| 2 | peter@peterson.com |
| 3 | lars@larson.eut |

**Membership** 表包含会员信息，比如用户是什么时候创建的，该会员是否已认证，会员是什么时候认证的，等等。

具体如下所示（一些列不显示）：

| User Id | Create Date | Confirmation Token | Is Confirmed | Last Password Failure | Password | F |
|---|---|---|---|---|---|---|
| 1 | 12.04.2012 16:12:17 | NULL | True | NULL | AFNQhWfy.... | 1 1 |

注释：如果您想看到所有的列和内容，请打开数据库，看看里边的每个表。

# 简单的会员配置

在您使用 WebSecurity 对象时，如果您的站点没有配置使用 ASP.NET Web Pages 会员系统 **SimpleMembership**，可能会报错。

如果托管服务提供商的服务器的配置与您本地服务器的配置不同，也可能会报错。为了解决这个问题，请在网站的 Web.config 文件中添加以下元素：

# ASP.NET Web Pages - Database 对象

## ASP.NET Database 对象参考手册

| 方法 | 描述 |
|---|---|
| Database.Execute(*SQLstatement [, parameters]*) | 执行 SQL 语句 *SQLstatement*（带可选参数），比如 INSERT、DELETE 或者 UPDATE， 并且返回受影响的记录统计。 |
| Database.GetLastInsertId() | 返回最近插入行的标识列。 |
| Database.Open(*filename*)<br>Database.Open(*connectionStringName*) | 使用 *Web.config* 文件中的连接字符串打开指定的数据库文件或者指定的数据库。 |
| Database.OpenConnectionString(*connectionString*) | 使用连接字符串打开一个数据库。（与 Database.Open 的差异是，Database.Open 使用的是连接字符串的名称，连接字符串的值在其他地方配置。） |
| Database.Query(*SQLstatement[, parameters]*) | 使用 SQL 语句 *SQLstatement*（带可选参数）查询数据库，并返回结果集合。 |
| Database.QuerySingle(*SQLstatement [, parameters]*) | 执行 SQL 语句 *SQLstatement*（带可选参数），并返回单条记录。 |
| Database.QueryValue(*SQLstatement [, parameters]*) | 执行 SQL 语句 *SQLstatement*（带可选参数），并返回单个值。 |

# ASP.NET Web Pages - WebMail 对象

通过 WebMail 对象，您可以很容易地从网页上发送电子邮件。

## 描述

**WebMail** 对象为 ASP.NET Web Pages 提供了使用 SMTP（Simple Mail Transfer Protocol 简单邮件传输协议）发送邮件的功能。

## 实例

请查看 WebPages Email 章节中的实例。

## WebMail 对象参考手册 - 属性

| 属性 | 描述 |
|---|---|
| SmtpServer | 用于发送电子邮件的 SMTP 服务器的名称。 |
| SmtpPort | 服务器用来发送 SMTP 电子邮件的端口。 |
| EnableSsl | 如果服务器使用 SSL（Secure Socket Layer 安全套接层）加密，则值为 true。 |
| UserName | 用于发送电子邮件的 SMTP 电子邮件账户的名称。 |
| Password | SMTP 电子邮件账户的密码。 |
| From | 在发件地址栏显示的电子邮件（通常与 UserName 相同）。 |

## WebMail 对象参考手册 - 方法

| 方法 | 描述 |
|---|---|
| Send() | 向 SMTP 服务器发送需要传送的电子邮件信息。 |

Send() 方法有以下参数：

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| to | String | 收件人（用分号分隔） |
| subject | String | 邮件主题 |
| body | String | 邮件正文 |

Send() 方法有以下可选参数：

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| from | String | 发件人 |
| cc | String | 需要抄送的电子邮件地址（用分号分隔） |
| filesToAttach | Collection | 附件名 |
| isBodyHtml | Boolean | 如果邮件正文是 HTML 格式的，则为 true |
| additionalHeaders | Collection | 附加的标题 |

# 技术数据

| 名称 | 值 |
| --- | --- |
| Class | System.Web.Helpers.WebMail |
| Namespace | System.Web.Helpers |
| Assembly | System.Web.Helpers.dll |

# 初始化 **WebMail** 帮助器

要使用 WebMail 帮助器，您必须能访问 SMTP 服务器。SMTP 是电子邮件的"输出"部分。如果您使用的是虚拟主机，您可能已经知道 SMTP 服务器的名称。如果您使用的是公司网络工作，您公司的 IT 部门会给您一个名称。如果您是在家工作，你也许可以使用普通的电子邮件服务提供商。

为了发送一封电子邮件，您将需要：

- SMTP 服务器的名称
- 端口号（通常是 25 ）
- 电子邮件的用户名
- 电子邮件的密码

在您的 Web 根目录下，创建一个名为 **_AppStart.cshtml** 的页面（如果已存在，则直接编辑页面）。

将下面的代码复制到文件中：

# _AppStart.cshtml

```
@{
WebMail.SmtpServer = "smtp.example.com";
WebMail.SmtpPort = 25;
WebMail.EnableSsl = false;
WebMail.UserName = "support@example.com";
WebMail.Password = "password";
WebMail.From = "john@example.com"
}
```

上面的代码将在每次网站（应用程序）启动时运行。它对 **WebMail** 对象赋了初始值。

请替换：

将 **smtp.example.com** 替换成您要用来发送电子邮件的 SMTP 服务器的名称。

将 **25** 替换成服务器用来发送 SMTP 事务（电子邮件）的端口号。

如果服务器使用 SSL（Secure Socket Layer 安全套接层）加密，请将 **false** 替换成 true。

将 **support@example.com** 替换成用来发送电子邮件的 SMTP 电子邮件账户的名称。

将 **password** 替换成 SMTP 电子邮件账户的密码。

将 **john@example** 替换成显示在发件地址栏中的电子邮件。

>  在您的 AppStart 文件中，您不需要启动 **WebMail** 对象，但是在调用 **WebMail.Send()** 方法之前，您必须设置这些属性。

# ASP.NET Web Pages - 更多帮助器

## ASP.NET 帮助器 - 对象参考手册

## Analytics 对象参考手册（Google）

| Helper | 描述 |
| --- | --- |
| Analytics.GetGoogleHtml(*webPropertyId*) | 为指定的 ID 呈现 Google Analytics JavaScript 代码。 |
| Analytics.GetStatCounterHtml(*project*, *security*) | 为指定的项目呈现 StatCounter Analytics JavaScript 代码。 |
| Analytics.GetYahooHtml(*account*) | 为指定的账号呈现 Yahoo Analytics JavaScript 代码。 |

## Bing 对象参考手册

| Helper | 描述 |
| --- | --- |
| Bing.SearchBox(*[boxWidth]*) | 给 Bing 传递搜索。您可以设置 Bing.SiteUrl 和 Bing.SiteTitle 属性来设定站点搜索和搜索框的标题，通常是在 _*AppStart* 页面设置这些属性。 |
| Bing.AdvancedSearchBox(*[, boxWidth] [, resultWidth] [, resultHeight] [, themeColor] [, locale]*) | 用可选的格式显示 Bing 搜索结果在页面上。您可以设置 Bing.SiteUrl 和 Bing.SiteTitle 属性来设定站点搜索和搜索框的标题，通常是在 _*AppStart* 页面设置这些属性。 |

## Chart 对象参考手册

| Helper | 描述 |
| --- | --- |
| Chart(*width*, *height [, template] [, templatePath]*) | 初始化图表。 |
| Chart.AddLegend(*[title] [, name]*) | 给图表添加一个图例。 |
| Chart.AddSeries(*[name] [, chartType] [, chartArea] [, axisLabel] [, legend] [, markerStep] [, xValue] [, xField] [, yValues] [, yFields] [, options]*) | 给图表添加一系列数据。 |

# Crypto 对象参考手册

| Helper | 描述 |
|---|---|
| Crypto.Hash(*string [, algorithm]*)<br>Crypto.Hash(*bytes [, algorithm]*) | 返回指定数据的哈希。默认算法是 sha256。 |

# Facebook 对象参考手册

| Helper | 描述 |
|---|---|
| Facebook.LikeButton(*href [, buttonLayout] [, showFaces] [, width] [, height] [, action] [, font] [, colorScheme] [, refLabel]*) | 让 Facebook 用户连接到网页。 |

# FileUpload 对象参考手册

| Helper | 描述 |
|---|---|
| FileUpload.GetHtml(*[initialNumberOfFiles] [, allowMoreFilesToBeAdded] [, includeFormTag] [, addText] [, uploadText]*) | 为上传文件呈现 UI。 |

# GamerCard 对象参考手册

| Helper | 描述 |
|---|---|
| GamerCard.GetHtml(*gamerTag*) | 呈现指定的 Xbox gamer 标签。 |

# Gravatar 对象参考手册

| Helper | 描述 |
|---|---|
| Gravatar.GetHtml(*email [, imageSize] [, defaultImage] [, rating] [, imageExtension] [, attributes]*) | 为指定的电子邮件地址呈现 Gravatar 图像。 |

# Json 对象参考手册

| Helper | 描述 |
|---|---|
| Json.Encode(*object*) | 用 JavaScript Object Notation (JSON) 把数据对象转换为字符串。 |
| Json.Decode(*string*) | 转换 JSON 编码的输入字符串为您指定的数据对象。 |

# LinkShare 对象参考手册

| Helper | 描述 |
| --- | --- |
| LinkShare.GetHtml(*pageTitle [, pageLinkBack] [, twitterUserName] [, additionalTweetText] [, linkSites]*) | 使用指定的标题和可选的 URL 呈现社会网络链接。 |

# ModelState 对象参考手册

| Helper | 描述 |
| --- | --- |
| ModelStateDictionary.AddError(*key*, *errorMessage*) | 关联错误信息和一个表单域。使用 ModelState 帮助器访问成员。 |
| ModelStateDictionary.AddFormError(*errorMessage*) | 关联错误信息和一个表单。使用 ModelState 帮助器访问成员。 |
| ModelStateDictionary.IsValid | 如果没有验证错误，返回 true。使用 ModelState 帮助器访问成员。 |

# ObjectInfo 对象参考手册

| Helper | 描述 |
| --- | --- |
| ObjectInfo.Print(*value [, depth] [, enumerationLength]*) | 呈现一个对象和所有子对象的属性和值。 |

# Recaptcha 对象参考手册

| Helper | 描述 |
| --- | --- |
| Recaptcha.GetHtml(*[, publicKey] [, theme] [, language] [, tabIndex]*) | 呈现 reCAPTCHA 验证测试。 |
| ReCaptcha.PublicKey<br>ReCaptcha.PrivateKey | 设置 reCAPTCHA 服务的公共和私有密钥。通常是在 *_AppStart* 页面设置这些属性。 |
| ReCaptcha.Validate(*[, privateKey]*) | 返回 reCAPTCHA 测试结果。 |
| ServerInfo.GetHtml() | Renders 呈现有关 ASP.NET Web Pages 的状态信息。 |

# Twitter 对象参考手册

| Helper | 描述 |
|---|---|
| Twitter.Profile(*twitterUserName*) | 为指定的用户呈现 Twitter 流。 |
| Twitter.Search(*searchQuery*) | 为指定的搜索文本呈现 Twitter 流。 |

# Video 对象参考手册

| Helper | 描述 |
|---|---|
| Video.Flash(*filename [, width, height]*) | 为指定的文件呈现宽度和高度可选的 Flash 视频播放。 |
| Video.MediaPlayer(*filename [, width, height]*) | 为指定的文件呈现宽度和高度可选 的 Windows Media 播放器。 |
| Video.Silverlight(*filename*, *width*, *height*) | 为指定的 *.xap* 文件呈现所需的宽度和高度 的 Silverlight 播放器。 |

# WebCache 对象参考手册

| Helper | 描述 |
|---|---|
| WebCache.Get(*key*) | 通过 *key* 返回指定的对象，如果对象未找到则返回 null。 |
| WebCache.Remove(*key*) | 通过 *key* 从缓存中删除指定的对象。 |
| WebCache.Set(*key*, *value [, minutesToCache] [, slidingExpiration]*) | 通过 *key* 把 *value* 放置到指定名称的缓存中。 |

# WebGrid 对象参考手册

| Helper | 描述 |
|---|---|
| WebGrid(*data*) | Creates a 使用查询数据创建一个新的 WebGrid 对象。 |
| WebGrid.GetHtml() | Renders markup 显示数据在 HTML 表格中。 |
| WebGrid.Pager() | 为 WebGrid 对象呈现一个页面。 |

# WebImage 对象参考手册

| Helper | 描述 |
|---|---|
| WebImage(*path*) | 从指定的路径加载一个图像。 |
| WebImage.AddImagesWatermark(*image*) | 为指定图像加水印。 |
| WebImage.AddTextWatermark(*text*) | 为图像添加指定文本。 |
| WebImage.FlipHorizontal()<br>WebImage.FlipVertical() | 水平/垂直翻转图像 |
| WebImage.GetImageFromRequest() | 当图像被传送到一个文件上传页面时，加载图像。 |
| WebImage.Resize(*width*, *height*) | 调整图像大小。 |
| WebImage.RotateLeft()<br>WebImage.RotateRight() | 向左或向右旋转图像。 |
| WebImage.Save(*path [, imageFormat]*) | 保存图像到指定路径。 |

# ASP.NET MVC - 参考手册

## 类

| 类 | 描述 |
|---|---|
| AcceptVerbsAttribute | 表示一个特性，该特性指定操作方法将响应的 HTTP 谓词。 |
| ActionDescriptor | 提供有关操作方法的信息，比如操作方法的名称、控制器、参数、特性和筛选器。 |
| ActionExecutedContext | 提供 ActionFilterAttribute 类的 ActionExecuted 方法的上下文。 |
| ActionExecutingContext | 提供 ActionFilterAttribute 类的 ActionExecuting 方法的上下文。 |
| ActionFilterAttribute | 表示筛选器特性的基类。 |
| ActionMethodSelectorAttribute | 表示一个用于影响操作方法选择的特性。 |
| ActionNameAttribute | 表示一个用于操作的名称的特性。 |
| ActionNameSelectorAttribute | 表示一个可影响操作方法选择的特性。 |
| ActionResult | 封装一个操作方法的结果并用于代表该操作方法执行框架级操作。 |
| AdditionalMetadataAttribute | 提供一个类，该类实现 IMetadataAware 接口以支持其他元数据。 |
| AjaxHelper | 表示支持在视图中呈现 AJAX 方案中的 HTML。 |
| AjaxHelper(TModel) | 表示支持在强类型视图中呈现 AJAX 方案中的 HTML。 |
| AjaxRequestExtensions | 表示一个类，该类对 HttpRequestBase 类进行了扩展，在其中添加了确定 HTTP 请求是否为 AJAX 请求的功能。 |
| AllowHtmlAttribute | 通过跳过属性的请求验证，允许请求在模型绑定过程中包含 HTML 标记。（强烈建议应用程序显式检查所有禁用请求验证的模型，以防止脚本攻击。） |
| AreaRegistration | 提供在一个 ASP.NET MVC 应用程序内注册一个或多个区域的方式。 |
| AreaRegistrationContext | 对在 ASP.NET MVC 应用程序内注册某个区域时所需的信息进行封装。 |
| AssociatedMetadataProvider | 提供用于实现元数据提供程序的抽象类。 |

| | |
|---|---|
| AssociatedValidatorProvider | 为用于实现验证提供程序的类提供抽象类。 |
| AsyncController | 为异步控制器提供基类。 |
| AsyncTimeoutAttribute | 表示一个特性，该特性用于设置异步方法的超时值（以毫秒为单位）。 |
| AuthorizationContext | 对使用 AuthorizeAttribute 特性时所需的信息进行封装。 |
| AuthorizeAttribute | 表示一个特性，该特性用于限制调用方对操作方法的访问。 |
| BindAttribute | 表示一个特性，该特性用于提供有关应如何进行模型绑定到参数的详细信息。 |
| BuildManagerCompiledView | 表示在视图引擎呈现视图之前由 BuildManager 类编译的视图的基类。 |
| BuildManagerViewEngine | 为视图引擎提供基类。 |
| ByteArrayModelBinder | 映射浏览器请求到字节数组。 |
| ChildActionOnlyAttribute | 表示一个特性，该特性用于指示操作方法只应作为子操作进行调用。 |
| ChildActionValueProvider | 表示子操作中的值的值提供程序。 |
| ChildActionValueProviderFactory | 表示用于为子操作创建值提供程序对象的工厂。 |
| ClientDataTypeModelValidatorProvider | 返回客户端数据类型模型验证程序。 |
| CompareAttribute | 提供用于比较某个模型的两个属性的特性。 |
| ContentResult | 表示用户定义的内容类型，该类型是操作方法的结果。 |
| Controller | 提供用于响应对 ASP.NET MVC 网站所进行的 HTTP 请求的方法。 |
| ControllerActionInvoker | 表示一个类，该类负责调用控制器的操作方法。 |
| ControllerBase | 表示所有 MVC 控制器的基类。 |
| ControllerBuilder | 表示一个类，该类负责动态生成控制器。 |
| ControllerContext | 封装有关与指定的 RouteBase 和 ControllerBase 实例匹配的 HTTP 请求的信息。 |
| ControllerDescriptor | 封装描述控制器的信息，比如控制器的名称、类型和操作。 |
| ControllerInstanceFilterProvider | 将控制器添加到 FilterProviderCollection 实例。 |

| | |
|---|---|
| CustomModelBinderAttribute | 表示一个调用自定义模型联编程序的特性。 |
| DataAnnotationsModelMetadata | 为数据模型的公共元数据、DataAnnotationsModelMetadataProvider 类和 DataAnnotationsModelValidator 类提供容器。 |
| DataAnnotationsModelMetadataProvider | 实现 ASP.NET MVC 的默认模型元数据提供程序。 |
| DataAnnotationsModelValidator | 提供模型验证程序。 |
| DataAnnotationsModelValidator(TAttribute) | 为指定的验证类型提供模型验证程序。 |
| DataAnnotationsModelValidatorProvider | 实现 ASP.NET MVC 的默认验证提供程序。 |
| DataErrorInfoModelValidatorProvider | 为错误信息模型验证程序提供容器。 |
| DefaultControllerFactory | 表示默认情况下已注册的控制器工厂。 |
| DefaultModelBinder | 映射浏览器请求到数据对象。该类提供模型联编程序的具体实现。 |
| DefaultViewLocationCache | 表示视图位置的内存缓存。 |
| DependencyResolver | 为实现 IDependencyResolver 或公共服务定位器 IServiceLocator 接口的依赖关系解析程序提供一个注册点。 |
| DependencyResolverExtensions | 提供 GetService 和 GetServices 的类型安全实现。 |
| DictionaryValueProvider(TValue) | 表示值提供程序的基类，这些值提供程序的值来自实现 IDictionary(TKey, TValue) 接口的集合。 |
| EmptyModelMetadataProvider | 为不需要元数据的数据模型提供空的元数据提供程序。 |
| EmptyModelValidatorProvider | 为不需要验证程序的模型提供空的验证提供程序。 |
| EmptyResult | 表示一个不执行任何操作的结果，比如一个不返回任何内容的控制器操作方法。 |
| ExceptionContext | P提供使用 HandleErrorAttribute 类的上下文。 |
| ExpressionHelper | 提供用于从表达式中获取模型名称的帮助器类。 |
| FieldValidationMetadata | 为客户端字段验证元数据提供容器。 |
| FileContentResult | 将二进制文件的内容发送到响应。 |
| FilePathResult | 将文件的内容发送到响应。 |

| | |
|---|---|
| FileResult | 表示一个用于将二进制文件内容发送到响应的基类。 |
| FileStreamResult | 使用 Stream 实例将二进制内容发送到响应。 |
| Filter | 表示一个元数据类，它包含对一个或多个筛选器接口的实现、筛选器顺序和筛选器范围的引用。 |
| FilterAttribute | 表示操作和结果筛选器特性的基类。 |
| FilterAttributeFilterProvider | 定义筛选器特性的筛选器提供程序。 |
| FilterInfo | 封装有关可用的操作筛选器的信息。 |
| FilterProviderCollection | 表示应用程序的筛选器提供程序的集合。 |
| FilterProviders | 为筛选器提供一个注册点。 |
| FormCollection | 包含应用程序的表单值提供程序。 |
| FormContext | 对验证和处理 HTML 表单中的输入数据所需的信息进行封装。 |
| FormValueProvider | 表示 NameValueCollection 对象中包含的表单值的值提供程序。 |
| FormValueProviderFactory | 表示一个类，该类负责创建表单值提供程序对象的新实例。 |
| GlobalFilterCollection | 表示一个包含所有全局筛选器的类。 |
| GlobalFilters | 表示全局筛选器集合。 |
| HandleErrorAttribute | 表示一个特性，该特性用于处理由操作方法引发的异常。 |
| HandleErrorInfo | 封装有关处理由操作方法引发的错误的信息。 |
| HiddenInputAttribute | 表示一个特性，该特性用于指示是否应将属性值或字段值呈现为隐藏的 input 元素。 |
| HtmlHelper | 表示支持在视图中呈现 HTML 控件。 |
| HtmlHelper(TModel) | 表示支持在强类型视图中呈现 HTML 控件。 |
| HttpDeleteAttribute | 表示一个特性，该特性用于限制操作方法，以便该方法仅处理 HTTP DELETE 请求。 |
| HttpFileCollectionValueProvider | 表示要用于来自 HTTP 文件集合的值的值提供程序。 |
| HttpFileCollectionValueProviderFactory | 表示一个类，该类负责创建 HTTP 文件集合值提供程序对象的新实例。 |

| | |
|---|---|
| HttpGetAttribute | 表示一个特性，该特性用于限制操作方法，以便该方法仅处理 HTTP GET 请求。 |
| HttpNotFoundResult | 定义一个用于指示未找到所请求资源的对象。 |
| HttpPostAttribute | 表示一个特性，该特性用于限制操作方法，以便该方法仅处理 HTTP POST 请求。 |
| HttpPostedFileBaseModelBinder | 将模型绑定到已发布的文件。 |
| HttpPutAttribute | 表示一个特性，该特性用于限制操作方法，以便该方法仅处理 HTTP PUT 请求。 |
| HttpRequestExtensions | 扩展 HttpRequestBase 类，该类包含客户端在 Web 请求中发送的 HTTP 值。 |
| HttpStatusCodeResult | 提供一种用于返回带特定 HTTP 响应状态代码和说明的操作结果的方法。 |
| HttpUnauthorizedResult | 表示未经授权的 HTTP 请求的结果。 |
| JavaScriptResult | 将 JavaScript 内容发送到响应。 |
| JsonResult | 表示一个类，该类用于将 JSON 格式的内容发送到响应。 |
| JsonValueProviderFactory | 启用操作方法以发送和接收 JSON 格式的文本，并将 JSON 文本以模型绑定方式传递给操作方法的参数。 |
| LinqBinaryModelBinder | 映射浏览器请求到 LINQ Binary 对象。 |
| ModelBinderAttribute | 表示一个特性，该特性用于将模型类型关联到模型-生成器类型。 |
| ModelBinderDictionary | 表示一个类，该类包含应用程序的所有模型联编程序（按联编程序类型列出）。 |
| ModelBinderProviderCollection | 为模型联编程序提供程序提供一个容器。 |
| ModelBinderProviders | 为模型联编程序提供程序提供一个容器。 |
| ModelBinders | 提供对应用程序的模型联编程序的全局访问。 |
| ModelBindingContext | 提供运行模型联编程序的上下文。 |
| ModelClientValidationEqualToRule | 为发送到浏览器的相等验证规则提供一个容器。 |
| ModelClientValidationRangeRule | 为发送到浏览器的范围验证规则提供一个容器。 |
| ModelClientValidationRegexRule | 为发送到浏览器的正则表达式客户端验证规则提供一个容器。 |
| ModelClientValidationRemoteRule | 为发送到浏览器的远程验证规则提供一个容器。 |

| | |
|---|---|
| ModelClientValidationRequiredRule | 为必填字段的客户端验证提供一个容器。 |
| ModelClientValidationRule | 为发送到浏览器的客户端验证规则提供一个基类容器。 |
| ModelClientValidationStringLengthRule | 为发送到浏览器的字符串长度验证规则提供一个容器。 |
| ModelError | 表示在模型绑定期间发生的错误。 |
| ModelErrorCollection | ModelError 实例的集合。 |
| ModelMetadata | 为数据模型的公共元数据、ModelMetadataProvider 类和 ModelValidator 类提供容器。 |
| ModelMetadataProvider | 为自定义元数据提供程序提供抽象基类。 |
| ModelMetadataProviders | 为当前的 ModelMetadataProvider 实例提供容器。 |
| ModelState | 将模型绑定的状态封装到操作方法参数的一个属性或操作方法参数本身。 |
| ModelStateDictionary | 表示将已发送表单绑定到操作方法（其中包括验证信息）的尝试的状态。 |
| ModelValidationResult | 为验证结果提供容器。 |
| ModelValidator | 提供用于实现验证逻辑的基类。 |
| ModelValidatorProvider | 为模型提供验证程序的列表。 |
| ModelValidatorProviderCollection | 为验证提供程序的列表提供一个容器。 |
| ModelValidatorProviders | 为当前验证提供程序提供容器。 |
| MultiSelectList | 表示一个项列表，用户可从该列表中选择多个项。 |
| MvcFilter | 在派生类中实现时，提供一个元数据类，它包含对一个或多个筛选器接口的实现、筛选器顺序和筛选器范围的引用。 |
| MvcHandler | 选择将处理 HTTP 请求的控制器。 |
| MvcHtmlString | 表示不应再次进行编码的 HTML 编码的字符串。 |
| MvcHttpHandler | 验证并处理 HTTP 请求。 |
| MvcRouteHandler | 创建一个实现 IHttpHandler 接口的对象并向该对象传递请求上下文。 |
| MvcWebRazorHostFactory | 创建 MvcWebPageRazorHost 文件的实例。 |
| NameValueCollectionExtensions | 扩展 NameValueCollection 对象，以便能够将集合复制到指定字典。 |

| | |
|---|---|
| NameValueCollectionValueProvider | 表示值提供程序的基类，这些值提供程序的值来自 NameValueCollection 对象。 |
| NoAsyncTimeoutAttribute | 为 AsyncTimeoutAttribute 特性提供便利包装。 |
| NonActionAttribute | 表示一个特性，该特性用于指示控制器方法不是操作方法。 |
| OutputCacheAttribute | 表示一个特性，该特性用于标记将缓存其输出的操作方法。 |
| ParameterBindingInfo | 封装与将操作方法参数绑定到数据模型相关的信息。 |
| ParameterDescriptor | 包含描述参数的信息。 |
| PartialViewResult | 表示一个用于将部分视图发送到响应的基类。 |
| PreApplicationStartCode | 为 ASP.NET Razor 应用程序预启动代码提供注册点。 |
| QueryStringValueProvider | 表示 NameValueCollection 对象中包含的查询字符串的值提供程序。 |
| QueryStringValueProviderFactory | 表示一个类，该类负责创建查询字符串值提供程序对象的新实例。 |
| RangeAttributeAdapter | 提供 RangeAttribute 特性的适配器。 |
| RazorView | 表示用于创建具有 Razor 语法的视图的类。 |
| RazorViewEngine | 表示一个用于呈现使用 ASP.NET Razor 语法的 Web 页面的视图引擎。 |
| RedirectResult | 通过重定向到指定的 URI 来控制对应用程序操作的处理。 |
| RedirectToRouteResult | 表示使用指定的路由值字典来执行重定向的结果。 |
| ReflectedActionDescriptor | 包含描述反射的操作方法的信息。 |
| ReflectedControllerDescriptor | 包含描述反射的控制器的信息。 |
| ReflectedParameterDescriptor | 包含描述反射的操作方法参数的信息。 |
| RegularExpressionAttributeAdapter | 提供 RegularExpressionAttribute 特性的适配器。 |
| RemoteAttribute | 提供使用 jQuery 验证插件远程验证程序的特性。 |
| RequiredAttributeAdapter | 提供 RequiredAttributeAttribute 特性的适配器。 |
| RequireHttpsAttribute | 表示一个特性，该特性用于强制通过 HTTPS 重新发送不安全的 HTTP 请求。 |

| | |
|---|---|
| ResultExecutedContext | 提供 ActionFilterAttribute 类的 OnResultExecuted 方法的上下文。 |
| ResultExecutingContext | 提供 ActionFilterAttribute 类的 OnResultExecuting 方法的上下文。 |
| RouteCollectionExtensions | 扩展 RouteCollection 对象以进行 MVC 路由。 |
| RouteDataValueProvider | 表示实现 IDictionary(TKey, TValue) 接口的对象中包含的路由数据的值提供程序。 |
| RouteDataValueProviderFactory | 表示用来创建路由数据值提供程序对象的工厂。 |
| SelectList | 表示一个列表，用户可从该列表中选择一个项。 |
| SelectListItem | 表示 SelectList 类的实例中的选定项。 |
| SessionStateAttribute | 指定控制器的会话状态。 |
| SessionStateTempDataProvider | 为当前 TempDataDictionary 对象提供会话状态数据。 |
| StringLengthAttributeAdapter | 提供 StringLengthAttribute 特性的适配器。 |
| TempDataDictionary | 表示仅从一个请求保持到下一个请求的数据集。 |
| TemplateInfo | 封装有关当前模板上下文的信息。 |
| UrlHelper | 包含用于为应用程序内的 ASP.NET MVC 生成 URL 的方法。 |
| UrlParameter | 表示路由过程中 MvcHandler 类使用的可选参数。 |
| ValidatableObjectAdapter | 提供可验证的对象适配器。 |
| ValidateAntiForgeryTokenAttribute | 表示用于阻止伪造请求的特性。 |
| ValidateInputAttribute | 表示一个特性，该特性用于标记必须验证其输入的操作方法。 |
| ValueProviderCollection | 表示应用程序的值提供程序对象的集合。 |
| ValueProviderDictionary | 已过时。表示应用程序的值提供程序的字典。 |
| ValueProviderFactories | 表示值提供程序工厂对象的容器。 |
| ValueProviderFactory | 表示用来创建值提供程序对象的工厂。 |
| ValueProviderFactoryCollection | 表示应用程序的值提供程序工厂的集合。 |
| ValueProviderResult | 表示将一个值（如表单发送的值或查询字符串中的值）绑定到操作方法参数属性或 |

| | |
|---|---|
| | 绑定到该参数本身的结果。 |
| ViewContext | 封装与呈现视图相关的信息。 |
| ViewDataDictionary | 表示一个容器，该容器用于在控制器和视图之间传递数据。 |
| ViewDataDictionary(TModel) | 表示一个容器，该容器用于在控制器和视图之间传递强类型数据。 |
| ViewDataInfo | 对开发模板所使用的当前模板内容和与模板交互的 HTML 帮助器的相关信息进行封装。 |
| ViewEngineCollection | 表示对应用程序可用的视图引擎的集合。 |
| ViewEngineResult | 表示定位视图引擎的结果。 |
| ViewEngines | 表示对应用程序可用的视图引擎的集合。 |
| ViewMasterPage | 表示生成母版视图页所需的信息。 |
| ViewMasterPage(TModel) | 表示生成强类型母版视图页所需的信息。 |
| ViewPage | 表示将视图呈现为 Web Forms 页所需的属性和方法。 |
| ViewPage(TModel) | 表示将强类型视图呈现为 Web Forms 页所需的信息。 |
| ViewResult | 表示一个类，该类用于使用由 IViewEngine 对象返回的 IView 实例来呈现视图。 |
| ViewResultBase | 表示一个用于为视图提供模型并向响应呈现视图的基类。 |
| ViewStartPage | 提供可用于实现视图启动（母版）页的抽象类。 |
| ViewTemplateUserControl | 提供 TemplateInfo 对象的容器。 |
| ViewTemplateUserControl(TModel) | 提供 TemplateInfo 对象的容器。 |
| ViewType | 表示视图的类型。 |
| ViewUserControl | 表示生成用户控件所需的信息。 |
| ViewUserControl(TModel) | 表示生成强类型用户控件所需的信息。 |
| VirtualPathProviderViewEngine | 表示 IViewEngine 接口的抽象基类实现。 |
| WebFormView | 表示在 ASP.NET MVC 中生成 Web Forms 页时所需的信息。 |
| WebFormViewEngine | 表示一个用于向响应呈现 Web Forms 页的视图引擎。 |
| WebViewPage | 表示呈现使用 ASP.NET Razor 语法的视图所需的属性和方法。 |

| WebViewPage(TModel) | 表示呈现使用 ASP.NET Razor 语法的视图所需的属性和方法。 |
| --- | --- |

# 接口

| 接口 | 描述 |
| --- | --- |
| IActionFilter | 定义操作筛选器中使用的方法。 |
| IActionInvoker | 定义操作调用程序的协定，该调用程序用于调用一个操作以响应 HTTP 请求。 |
| IAuthorizationFilter | 定义授权筛选器所需的方法。 |
| IClientValidatable | 为 ASP.NET MVC 验证框架提供一种用于在运行时发现验证程序是否支持客户端验证的方法。 |
| IController | 定义控制器所需的方法。 |
| IControllerActivator | 对使用依赖关系注入来实例化控制器的方式进行精细控制。 |
| IControllerFactory | 定义控制器工厂所需的方法。 |
| IDependencyResolver | 定义可简化服务位置和依赖关系解析的方法。 |
| IExceptionFilter | 定义异常筛选器所需的方法。 |
| IFilterProvider | 提供用于查找筛选器的接口。 |
| IMetadataAware | 提供用于向 AssociatedMetadataProvider 类公开特性的接口。 |
| IModelBinder | 定义模型联编程序所需的方法。 |
| IModelBinderProvider | 定义用于为实现 IModelBinder 接口的类动态实现模型绑定的方法。 |
| IMvcFilter | 定义用于指定筛选器顺序以及是否允许多个筛选器的成员。 |
| IResultFilter | 定义结果筛选器所需的方法。 |
| IRouteWithArea | 将路由与 ASP.NET MVC 应用程序中的区域关联。 |
| ITempDataProvider | 定义临时数据提供程序的协定，这些临时数据提供程序用于存储要在下一个请求中查看的数据。 |
| IUnvalidatedValueProvider | 表示一个可跳过请求验证的 IValueProvider 接口。 |
| IValueProvider | 定义 ASP.NET MVC 中的值提供程序所需的方法。 |
| IView | 定义视图所需的方法。 |
| IViewDataContainer | 定义视图数据字典所需的方法。 |
| IViewEngine | 定义视图引擎所需的方法。 |
| IViewLocationCache | 定义在内存中缓存视图位置所需的方法。 |
| IViewPageActivator | 对使用依赖关系注入创建视图页的方式进行精细控制。 |

# Web Forms 参考手册

# ASP.NET Web Forms - HTML 服务器控件

HTML 服务器控件是服务器可理解的 HTML 标签。

## HTML 服务器控件

ASP.NET 文件中的 HTML 元素，默认是作为文本进行处理的。要想让这些元素可编程，需向 HTML 元素中添加 runat="server" 属性。这个属性表示，该元素将被作为服务器控件进行处理。

注释：所有 HTML 服务器控件必须位于带有 runat="server" 属性的 <form> 标签内！

注释：ASP.NET 要求所有 HTML 元素必须正确关闭和正确嵌套。

| HTML 服务器控件 | 描述 |
|---|---|
| HtmlAnchor | 控制 `<a>` HTML 元素 |
| HtmlButton | 控制 `<button>` HTML 元素 |
| HtmlForm | 控制 `<form>` HTML 元素 |
| HtmlGeneric | 控制其他未被具体的 HTML 服务器控件规定的 HTML 元素，比如 `<body>` 、 `<div>` 、 `<span>` 等。 |
| HtmlImage | 控制 `<image>` HTML 元素 |
| HtmlInputButton | 控制 `<input type="button">` 、 `<input type="submit">` 和 `<input type="reset">` HTML 元素 |
| HtmlInputCheckBox | 控制 `<input type="checkbox">` HTML 元素 |
| HtmlInputFile | 控制 `<input type="file">` HTML 元素 |
| HtmlInputHidden | 控制 `<input type="hidden">` HTML 元素 |
| HtmlInputImage | 控制 `<input type="image">` HTML 元素 |
| HtmlInputRadioButton | 控制 `<input type="radio">` HTML 元素 |
| HtmlInputText | 控制 `<input type="text">` 和 `<input type="password">` HTML 元素 |
| HtmlSelect | 控制 `<select>` HTML 元素 |
| HtmlTable | 控制 `<table>` HTML 元素 |
| HtmlTableCell | 控制 `<td>` 和 `<th>` HTML 元素 |
| HtmlTableRow | 控制 `<tr>` HTML 元素 |
| HtmlTextArea | 控制 `<textarea>` HTML 元素 |

# ASP.NET Web Forms - Web 服务器控件

Web 服务器控件是服务器可理解的特殊 ASP.NET 标签。

## Web 服务器控件

就像 HTML 服务器控件，Web 服务器控件也是在服务器上创建的，它们同样需要 runat="server" 属性才能生效。然而，Web 服务器控件没有必要映射任何已存在的 HTML 元素，它们可以表示更复杂的元素。

创建 Web 服务器控件的语法是：

```
<asp:control_name id="some_id" runat="server" />
```

| Web 服务器控件 | 描述 |
|---|---|
| AdRotator | 显示一个图形序列 |
| Button | 显示下压按钮 |
| Calendar | 显示日历 |
| CalendarDay | calendar 控件中的一天 |
| CheckBox | 显示复选框 |
| CheckBoxList | 创建多选的复选框组 |
| DataGrid | 显示 grid 中数据源的字段 |
| DataList | 通过使用模版显示数据源中的项目 |
| DropDownList | 创建下拉列表 |
| HyperLink | 创建超链接 |
| Image | 显示图像 |
| ImageButton | 显示可点击的图像 |
| Label | 显示可编程的静态内容（使您对其内容应用样式） |
| LinkButton | 创建超链接按钮 |
| ListBox | 创建单选或多选的下拉列表 |
| ListItem | 创建列表中的一个项目 |
| Literal | 显示可编程的静态内容（无法使您对其内容应用样式） |
| Panel | 为其他控件提供容器 |
| PlaceHolder | 为由代码添加的控件预留空间 |
| RadioButton | 创建单选按钮 |
| RadioButtonList | 创建单选按钮组 |
| BulletedList | 创建项目符号格式的列表 |
| Repeater | 显示绑定到控件的项目的重复列表 |
| Style | 设置控件的样式 |
| Table | 创建表格 |
| TableCell | 创建表格单元格 |
| TableRow | 创建表格行 |
| TextBox | 创建文本框 |
| Xml | 显示 XML 文件或 XSL 转换的结果 |

# ASP.NET Web Forms - Validation 服务器控件

Validation 服务器控件是用来验证用户输入的。

## Validation 服务器控件

Validation 服务器控件用于验证输入控件的数据。如果数据未通过验证，则向用户显示错误消息。

创建 Validation 服务器控件的语法是：

```
<asp:control_name id="some_id" runat="server" />
```

| Validation 服务器控件 | 描述 |
| --- | --- |
| CompareValidator | 把一个输入控件的值与另一个输入控件的值或一个固定的值进行对比 |
| CustomValidator | 允许您编写一个方法，来处理输入值的验证 |
| RangeValidator | 检查用户输入值是否介于两个值之间 |
| RegularExpressionValidator | 确保输入控件的值匹配指定的模式 |
| RequiredFieldValidator | 使输入控件成为必需（必填）的字段 |
| ValidationSummary | 显示网页中所有验证错误的报告 |

# 免责声明